Handling Scalable Approximate Queries over NoSQL Graph Databases: Cypherf and the Fuzzy4S Framework

Arnaud Castelltort[1], Trevor Martin[2]

[1] *LIRMM, CNRS-University of Montpellier, France*
[2] *Department of Engineering Mathematics, University of Bristol, UK*

Abstract

NoSQL databases are currently often considered for Big Data solutions as they offer efficient solutions for volume and velocity issues and can manage some of complex data (e.g., documents, graphs). Fuzzy approaches are yet often not efficient on such frameworks. Thus this article introduces a novel approach to define and run approximate queries over NoSQL graph databases using Scala by proposing the Fuzzy4S framework and the Cypherf fuzzy declarative query language. NoSQL Graph databases are currently gaining more and more interest and are applied in many real world applications. The Fuzzy4S framework is defined with an open DSL (Domain Specific Language) allowing it to define scalable approximate queries at an abstract level. Cypherf is an extension of Cypher which runs over the Neo4J NoSQL graph databases. This work consists of a complete approach embedding the whole chain from end-user declarative query level to implementation issues within the database engine. We provide both the formal definitions for defining approximate graph NoSQL queries and the experimental results which demonstrate the interest and efficiency of our proposition.

Keywords: Approximate Queries, Fuzzy Logic, NoSQL Graph Databases.

1. Introduction

Graphs are often a natural way to represent the world [1]. This is especially true for social networks, but also in biology, chemistry etc. In such areas,

relationships between objects are at least as important as the objects themselves. For example, retrieving "friend" and "friend of friend" relationships is a key operation in social networks or in any other human relations. In [2], we have highlighted the power of the study of relations in order to retrieve fraud rings which are defined as *sophisticated chains of indirect links between fraudsters representing successive transactions (money, communications, etc.) from which rogue behaviours are detected*.

Graphs have been extensively studied in the literature and have recently gained attention with the development of Semantic Web and ontologies. Many modelizations, tools and frameworks have been suggested to represent, manage and analyze graphs with XML, OWL etc.

At the same time, the volume and complexity of information concerning these objects and their relationships are growing dramatically, leading to huge databanks.

This has led to the development of the so-called NoSQL graph databases that embed both the performance of NoSQL databases and the representativity of graphs. Several engines exist (OrientDB, Neo4J, HyperGraphDB etc.) [3]. In this article, Neo4J is used since it has been considered the top graph database management for several years [1] (Figure 1, [4][2]).

Such database management systems embed query processing that can be used at different levels, from the declarative level to the programmatic level within the engine. The Neo4J system offers the declarative Cypher language that allows the users to query the graph in a very intuitive manner. These queries include clauses over both nodes and relationships. Properties can be defined in the same style that NoSQL databases *i.e.* with $(key : value)$ lists. The value can be of any type, including collections.

For instance, it is possible to consider nodes representing people having some properties such as age ($key = age$, $value = 28$) or hobbies ($key = hobby$,

[1] http://db-engines.com/en/ranking/graph+dbms

[2] `http://thoughtworks.fileburst.com/assets/technology-radar-may-2013.pdf`, 2013

27 systems in ranking, July 2017

Rank Jul 2017	Rank Jun 2017	Rank Jul 2016	DBMS	Database Model	Score Jul 2017	Score Jun 2017	Score Jul 2016
1.	1.	1.	Neo4j	Graph DBMS	38.52	+0.65	+4.83
2.	2.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	7.71	+1.33	+5.29
3.	3.	↓ 2.	OrientDB	Multi-model	5.57	-0.24	-0.26
4.	4.	↓ 3.	Titan	Graph DBMS	4.93	-0.33	-0.34
5.	5.	↑ 6.	ArangoDB	Multi-model	2.96	-0.12	+1.21
6.	6.	↓ 5.	Virtuoso	Multi-model	1.99	-0.01	-0.31
7.	7.	7.	Giraph	Graph DBMS	1.05	+0.00	+0.16
8.	8.	8.	AllegroGraph	Multi-model	0.61	+0.00	+0.14
9.	9.	9.	Stardog	Multi-model	0.55	+0.01	+0.09
10.	10.	↑ 12.	GraphDB	Multi-model	0.53	+0.01	+0.36
11.	11.	↓ 10.	Sqrrl	Multi-model	0.49	+0.02	+0.24
12.	↑ 16.		Graph Engine	Multi-model	0.36	+0.10	
13.	13.	↓ 11.	InfiniteGraph	Graph DBMS	0.29	-0.01	+0.10
14.	↑ 15.	↑ 18.	Blazegraph	Multi-model	0.29	-0.00	+0.21
15.	↓ 14.		Dgraph	Graph DBMS	0.28	-0.02	
16.	↑ 19.		JanusGraph	Graph DBMS	0.23	+0.11	
17.	↑ 21.	↓ 16.	Sparksee	Graph DBMS	0.16	+0.05	+0.07
18.	↓ 17.	↓ 14.	FlockDB	Graph DBMS	0.16	-0.03	+0.03
19.	↓ 18.	↓ 17.	HyperGraphDB	Graph DBMS	0.15	-0.03	+0.07
20.	20.	↓ 15.	InfoGrid	Graph DBMS	0.13	+0.01	+0.04

Figure 1: Comparison of Graph Database Management Systems (db-engines.com)

$value = \{tennis, reading, travel\}$).

In some cases, it is not relevant for users to define queries in a “crisp" way, as for instance to retrieve people having friends aged between 20 and 30 but they rather want to retrieve people even if their age is 31 and not strictly between 20 and 30.

For this reason, we consider approximate queries over such numerical properties. Such queries can for instance be expressed using the fuzzy set theory framework. In our proposition we introduce Fuzzy4S, standing for *Fuzzy for Scala*. Scala is a modern language mixing object-oriented and functional programming [5]. Fuzzy4S is used within NoSQL databases to define fuzzy queries by introducing the Cypherf framework (which stands for *Cypher fuzzy*).

The rest of this article is organized as follows: Section 2 presents the preliminary concepts underlying our work: graph databases and approximate queries. Section 3 introduces the extension of graph NoSQL queries with the Cypherf framework and discusses implementation issues. Cypherf relies on the Fuzzy4S framework which is detailed in Section 4. Section 5 illustrates our contributions

using examples and showing the efficiency of our implementation with experimental load tests over benchmarks. Section 6 concludes this paper and discusses some of the many research avenues opened by our work.

2. Preliminary Concepts and Properties

This work merges two topics, namely graph oriented NoSQL databases and approximate queries that are briefly introduced below.

For illustrating these concepts and in the rest of this paper, we rely on an example where customers are linked to the hotels they have visited. A hotel is described with properties such as the price and location with regards to the city center. Customers are described by their age.

2.1. Graph Databases

Graphs have been studied for a long time by mathematicians and computer scientists. A graph can be directed or not, labeled or not. Graph databases rely on labeled directed graphs.

Definition 1 (Labeled Directed Graph). *A labeled oriented graph G, also known as oriented property graph, is given by a n-tuple $(V, E, \alpha, \beta, l_V, l_E)$ where V stands for a set of nodes and E stands for a set of edges with $E \subseteq (V \times V)$, α stands for the set of attributes defined over the nodes, β the set of attributes defined over the relations, $l_V : V \to \mathcal{P}(\alpha)$ is the labeling function for nodes and $l_E : E \to \mathcal{P}(\beta)$ is the labeling function for relationships.*

In such databases, α and β are not simple labels such as in labeled graphs. They are rather defined by $(key, value)$ pairs commonly used by NoSQL databases [6] to represent the so-called properties over nodes and relations. Keys can occur in several nodes or relations.

Figure 2 shows a graph and its anatomy in $(key, value)$ pairs.

Queries in graph databases are defined as traversals over the graph. They can be run at several levels, either programmatically or by using declarative

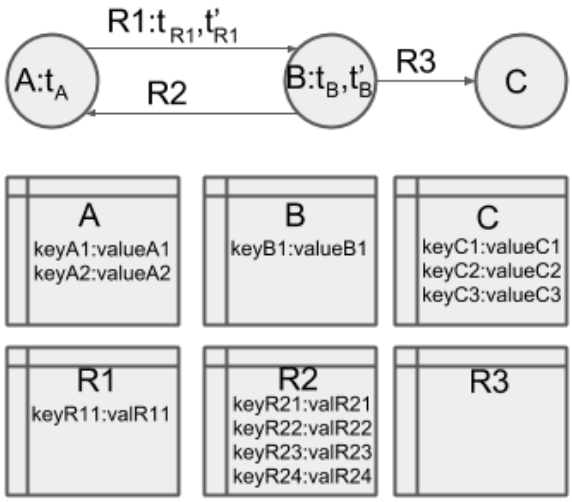


Figure 2: Node and Relationship Properties

languages (as done when using SQL in relational databases). No common set of operators has yet been defined as a standard. It should be noted that the result of a query over a graph is not always defined as a graph.

In Neo4J, declarative queries are run using the so-called *Cypher* language.

2.2. Neo4J Cypher Language

In Cypher, queries are formulated using the following syntax[3]:

```
[MATCH]
[OPTIONAL MATCH WHERE]
[WITH [ORDER BY] [SKIP] [LIMIT]]
RETURN [ORDER BY] [SKIP] [LIMIT]
```

Queries are formalized through EBNF Cypher grammar, as shown below. For the sake of simplicity, we only show some extracts of the grammar.

[3]http://neo4j.com/docs/2.2.5/cypher-refcard/
http://neo4j.com/docs/2.2.5/cypher-query-lang.html

An EBNF grammar [7] is an extension of BNF (Backus-Naur Form (BNF) [8]) for formalizing the syntax of a language. It has been extensively studied in software engineering research works. Roughly speaking, every EBNF grammar is described by means of rules, every rule being defined with a left and right part.

An open community works on the openCypher project[4] that aims to deliver a full and open specification of Cypher and to evolve the language through the production and acceptance of CIPs (Cypher Improvement Proposals). Our work is part of this trend and aims to contribute with fuzzy extensions.

EBNF Grammar Extract 1. *Cypher requests are based on sequences of statements that are regular queries. Regular queries are based on single queries that can be combined through unions. Single queries are based on 9 possible clauses: Match, Unwind, Merge, Create, Set, Delete, Remove, With, Return.*

```
Cypher = [SP], Statement, [[SP], ';'], [SP] ;

Statement = Query ;

Query = RegularQuery ;

RegularQuery = SingleQuery, { [SP], Union } ;

SingleQuery = Clause, { [SP], Clause } ;

Clause = Match
       | Unwind
       | Merge
       | Create
       | Set
       | Delete
```

[4] www.opencypher.org

```
        | Remove
        | With
        | Return
        ;

SP = { whitespace }- ;
```

As shown here, Cypher is comprised of several distinct clauses:

- MATCH: the graph pattern to match. The $MATCH$ clause is the primary way of getting data from the database into the current set of bindings by providing an example. It allows users to specify the patterns that Cypher will search for in the database.
- UNWIND: expands a list into a sequence of rows.
- MERGE: ensures that a pattern exists in the graph. Either the pattern already exists, or it is created.
- CREATE: creates nodes and relationships.
- SET: is used to update labels on nodes and properties on nodes and relationships.
- DELETE: removes graphs elements (nodes, relationships and paths).
- REMOVE: is used to remove properties and labels from graph elements.
- RETURN: what to return. It defines which parts of the pattern the user is interested in. It can be nodes, relationships, or properties on them.
- WITH: allows query parts to be chained together, piping the results from one to be used as starting points or criteria in the next.

For instance, it is possible to retrieve which hotels have been visited by customers between the ages of 20 and 30 as shown in listing 1.

Listing 1: Retrieve which hotels have been visited by customers between the ages of 20 and 30 Query

```
MATCH (hotel:Hotel)<-[:VISITED]-(c: Customer)
WHERE c.age <= 30 and c.age >=20
RETURN DISTINCT hotel
```

Such queries allow the user to answer many questions in a very efficient way. However, they do not allow soft queries, such as retrieving the hotels visited by *young* customers where *young* is defined as a fuzzy set over the numerical universe of all possible ages. For this reason, we claim that introducing approximate queries in NoSQL graph databases is important. The next section introduces approximate queries.

2.3. Approximate Queries

Many works have been proposed to deal with approximate data and queries. It should first be noted that the consideration of imperfection may have several semantics, should the data be uncertain, imprecise, vague, incomplete and/or inconsistent [9].

This work focuses on approximate queries considered as imprecise queries over precise (classical) data, relying on fuzzy set theory [10].

2.3.1. Fuzzy Queries

The works on fuzzy databases have dealt with the various topics: entity-relationship models, functional dependencies, relational models, object databases, queries, etc. We focus here on fuzzy queries over classical databases.

In [11, 12, 13] querying regular databases is considered by extending the SQL language.The FSQL/SQLf and FQL languages have been proposed to extend queries over relational databases in order to incorporate fuzzy descriptions of the information being searched for.

In such systems, approximation is basically associated to fuzzy labels, fuzzy comparators (e.g., *fuzzy greater than*) and aggregation over clauses. Thresholds can be defined for the expected fulfillment of fuzzy clauses.

For instance, when considering a database containing crisp descriptions of hotels, users can ask for *cheap* hotels that are *close to city center*, *cheap* and *close_to_city_center* being fuzzy labels described by fuzzy sets and their membership functions respectively defined on the universe of prices and distance to the city center.

For instance, SQLf allows to integrate fuzzy predicates within the $WHERE$ clause of an SQL query:

Listing 2: SQLf Query

```
SELECT *
FROM Table_Hotel
WHERE Price is cheap;
```

Some works have been implemented as fuzzy database engines with fuzzy querying features [14, 15].

Many works have been proposed to investigate how such fuzzy predicates can be defined by users and computed by the database engine, especially when several clauses must be merged. For instance, for retrieving a hotel that is both *cheap AND close to city center*, the membership degrees are aggregated with a t-norm.

Such aggregation can consider preferences, for instance for queries where price is preferred to distance to city center using weighted t-norms.

Thresholds can be added when searching for hotels where the *cheap* degree is greater than 0.7 for instance.

More detailed definitions of the formalism are provided in the next section.

As we consider graph data, the works on fuzzy ontology querying are very close and relevant [16, 17]. f-SPARQL provides a flexible extension for SPARQL by allowing users to apply fuzzy filters and fuzzy operators. Figure 3 shows the syntax of this extension.

However, no model has been proposed for NoSQL graph databases. We thus propose to extend our first work [18] for defining approximate queries over NoSQL graph databases, as described in the next section.

Query	::= Prologue (*QueryType* SelectQuery | ConstructQuery | DescribeQuery| AskQuery)
QueryType	::= '#FQ#' | #top-k FQ# with k
Constraint	::= BrackettedExpression | BuiltInCall | FunctionCall | FlexibleExpression
FlexibleExpression	::= FuzzyTermExpression | FuzzyOperatorExpression
FuzzyTermExpression	::= (Var ['=','!=','>''>=','<','<='] FuzzyTerm)? [with threshold]
FuzzyOperatorExpression	::= Var FuzzyOperator NumericLiteral
FuzzyOperator	::= (Modifier)*FuzzyOperator
FuzzyTerm	::= FuzzyTerm and FuzzyTerm | FuzzyTerm or FuzzyTerm | not FuzzyTerm | ModifiedFuzzyTerm
ModifiedFuzzyTerm	::=(Modifier)* ModifiedFuzzyTerm | (Modifier)* SimpleFuzzyTerm

Figure 3: f-SPARQL modifications to the SPARQL standard grammar

3. Fuzzy Queries over NoSQL Graph databases: Towards the Cypherf Language

3.1. Fuzzy Queries over Graph Dababases

This section addresses fuzzy queries over regular NoSQL Neo4J graph databases. We claim that fuzziness can be handled at the following three levels: over nodes, over properties, and over relationships.

The rest of this section presents each possibility along with its associated EBNF grammar.

EBNF Grammar Extract 2. *An expression is defined with sub-expressions that can be chained with logical and arithmetical operators.*

```
Expression = Expression12 ;

Expression12 = Expression11, { SP, (O,R), SP, Expression11 } ;

Expression11 = Expression10, { SP, (X,O,R), SP, Expression10 } ;

Expression10 = Expression9, { SP, (A,N,D), SP, Expression9 } ;

Expression9 = { (N,O,T), [SP] }, Expression8 ;

Expression8 = Expression7, { [SP], PartialComparisonExpression } ;
```

```
Expression7 = Expression6, { ([SP], '+', [SP], Expression6) | ([SP], '-', [SP], Expression6) } ;

Expression6 = Expression5, { ([SP], '*', [SP], Expression5) |
([SP], '/', [SP], Expression5) | ([SP], '%', [SP], Expression5) } ;

Expression5 = Expression4, { [SP], '^', [SP], Expression4 } ;

Expression4 = { ('+' | '-'), [SP] }, Expression3 ;

Expression3 = Expression2, { ([SP], '[', Expression, ']') |
([SP], '[', [Expression], '..', [Expression], ']') | (((SP], '=~') |
(SP, (I,N)) | (SP, (S,T,A,R,T,S), SP, (W,I,T,H)) |
(SP, (E,N,D,S), SP, (W,I,T,H)) | (SP, (C,O,N,T,A,I,N,S))), [SP], Expression2) |
(SP, (I,S), SP, (N,U,L,L)) |
(SP, (I,S), SP, (N,O,T), SP, (N,U,L,L)) } ;

Expression2 = Atom, { [SP], (PropertyLookup | NodeLabels) } ;

Atom = Literal
     | Parameter
     | ((C,O,U,N,T), [SP], '(', [SP], '*', [SP], ')')
     | ListComprehension
     | PatternComprehension
     | ((F,I,L,T,E,R), [SP], '(', [SP], FilterExpression, [SP], ')')
     | ((E,X,T,R,A,C,T), [SP], '(', [SP], FilterExpression, [SP], [[SP], '|', Expression], ')')
     | ((A,L,L), [SP], '(', [SP], FilterExpression, [SP], ')')
     | ((A,N,Y), [SP], '(', [SP], FilterExpression, [SP], ')')
     | ((N,O,N,E), [SP], '(', [SP], FilterExpression, [SP], ')')
     | ((S,I,N,G,L,E), [SP], '(', [SP], FilterExpression, [SP], ')')
     | RelationshipsPattern
     | ParenthesizedExpression
     | FunctionInvocation
```

```
        | Variable
        ;
```

3.1.1. Cypherf over Nodes

Dealing with fuzzy queries over nodes allows the user to retrieve similar nodes.

To achieve that goal, we decided to reuse the grammar definition available in Cypher. Indeed, we decided to use fuzzy functions that take nodes as parameters. Cypher supports function invocations in place of Expression.

EBNF Grammar Extract 3. *FunctionInvocation has a function name followed by zero to any numbers of parameters that are separated by commas.*

```
FunctionInvocation = FunctionName, [SP], '(', [SP], [(D,I,S,T,I,N,C,T), [SP]],
    [Expression, [SP], { ',', [SP], Expression, [SP] }], ')' ;
```

For instance, it is possible to retrieve similar hotels:

Listing 3: Getting Similar Hotel Nodes

```
1 MATCH (h1:Hotel),(h2:Hotel)
2 WITH h1 AS hot1, h2 AS hot2, SimilarTo(hot1,hot2) AS sim
3 WHERE sim > 0.7
4 RETURN hot1,hot2,sim
```

In this framework, the similarity between nodes is based on the definition of measures that merge the similarities of every property from the two nodes being compared. For example, a similarity measure can be built to compare and rank hotels based on the similarities of both price and size [19].

3.1.2. Cypherf over Properties

Dealing with fuzzy queries over properties leads to the consideration of fuzzy linguistic terms and/or fuzzy comparators.

```
MATCH (h:Hotel)-[:LOCATED]->(c:City)
RETURN h, CLOSE(h,c) AS ClosenessToCityCenter
ORDER BY ClosenessToCityCenter DESC
```

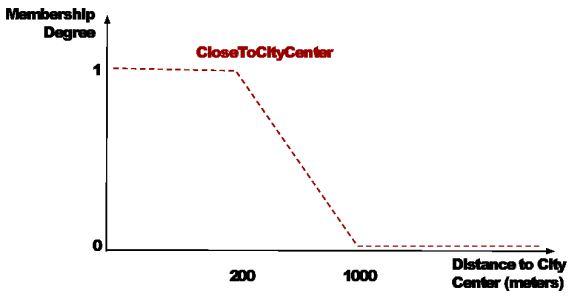


Figure 4: Proximity to City Center

Such fuzzy queries impact the $MATCH$, $WHERE$, $WITH$ and $RETURN$ clauses from Cypher.

In the $WHERE$ clause, it is then possible to search for *cheap* hotels in some databases, or for hotels located *close to city center*. Note that the properties being considered can be defined at the node and relationship levels.

Listing 4: Cheap Hotels

```
MATCH (h:Hotel)
WHERE CHEAP(price) > 0
RETURN h
ORDER BY CHEAP(h) DESC
```

Listing 5: Hotels Close to City Center

```
MATCH (c:City)<-[:LOCATED]-(h:Hotel)
WHERE CLOSE(c,h) > 0
RETURN h
ORDER BY CLOSE(c,h) DESC
```

In the $MATCH$ clause, integrating fuzzy criteria are also possible:

Listing 6: Matching Hotels Close to City Center

```
MATCH (h:Hotel)-[:LOCATED {CLOSE(distance)>0}]->(c:City)
RETURN h
ORDER BY CLOSE(h,c) DESC
```

To offer this possibility we have added the concept of fuzzy criterion to Cypher and we have extended its grammar. Indeed, Cypher is not able to use functions in "RelationshipPattern". We thus introduce the following extension.

EBNF Grammar Extension 1. *"FuzzyCriteria" is defined as a function invocation that is an optional member of a "RelationshipDetail". A Relationship detail is an optional part of a "RelationshipPattern" which is a part of a "PatternElementChain", and so on to the "Pattern".*

The extension of the RelationshipDetail possibilities by the addition of the FuzzyCriteria offers by transitivity the possibilitiy to the Match clause to use FuzzyCriteria in a relationship construction.

```
Match = [(O,P,T,I,O,N,A,L), SP], (MATCH), [SP], Pattern, [[SP], Where] ;

Merge = (M,E,R,G,E), [SP], PatternPart, { SP, MergeAction } ;

Create = (C,R,E,A,T,E), [SP], Pattern ;

Pattern = PatternPart, { [SP], ',', [SP], PatternPart } ;

PatternPart = (Variable, [SP], '=', [SP], AnonymousPatternPart)
            | AnonymousPatternPart
            ;

AnonymousPatternPart = PatternElement ;

PatternElement = (NodePattern, { [SP], PatternElementChain })
               | ('(', PatternElement, ')')
               ;
```

```
NodePattern = '(', [SP], [Variable, [SP]], [NodeLabels, [SP]], [Properties, [SP]], ')' ;

PatternElementChain = RelationshipPattern, [SP], NodePattern ;

RelationshipPattern = (LeftArrowHead, [SP], Dash, [SP], [RelationshipDetail],
                      [SP], Dash, [SP], RightArrowHead)
                      | (LeftArrowHead, [SP], Dash, [SP], [RelationshipDetail], [SP], Dash)
                      | (Dash, [SP], [RelationshipDetail], [SP], Dash, [SP], RightArrowHead)
                      | (Dash, [SP], [RelationshipDetail], [SP], Dash)
                      ;

RelationshipDetail = '[', [SP], [Variable, [SP]], [RelationshipTypes, [SP]],
[RangeLiteral], [FuzzyCriteria], [Properties, [SP]], ']' ;

FuzzyCriteria = '{' , [SP], FunctionInvocation, [SP], '}'
```

In the *RETURN* clause, no selection will be achieved, but fuzzy terms can be added in order to show the users the degree to which some values possess properties represented by fuzzy sets, as for instance:

Listing 7: Fuzziness in the Return Clause

```
MATCH (h:Hotel)-[:LOCATED]->(c:City)
RETURN h, CLOSE(h,c) AS 'ClosenessToCityCenter'
ORDER BY ClosenessToCityCenter DESC
```

3.1.3. Cypherf over Relationships

As for nodes, such queries may be based on properties. However it can also be based on the graph structure in order to better exploit and benefit from it.

In Cypher, the structure of the pattern being searched is mostly defined in the *MATCH* clause.

The first attempt to extend pattern matching to fuzzy pattern matching is to consider chains and depth matching. Chains are defined in Cypher in the

MATCH clause with consecutive links between objects. If a node a is linked to an object b at depth 2, the pattern is writen as $(a) - [*2] - > (b)$. If a link between a and b without regarding the depth in-between is searched, then it is written $(a) - [*] - > (b)$. The mechanism also applies for searching objects linked through a range of nodes (e.g. between 3 and 5): $(a) - [*3..5] - > (b)$.

In a similar way to [20], we propose thus to introduce fuzzy descriptors to define extended patterns where the depth is imprecisely described. It will then for instance be possible to search for customers linked through *almost 3* hops. The syntax $**$ is proposed to indicate a fuzzy linker.

EBNF Grammar Extension 2. *FuzzyLinker is defined as a FunctionInvocation that can be used in RelationshipDetail.*

```
RelationshipDetail = '[', [SP], [Variable, [SP]],
    [RelationshipTypes, [SP]], [RangeLiteral], [FuzzyLinker],
    [FuzzyCriteria], [Properties, [SP]], ']' ;

FuzzyLinker = '**', [SP], FunctionInvocation ;
```

Listing 8: Fuzzy Patterns

```
1 MATCH (c1:customer)-[:KNOWS**almost(3)]->(c2:customer)
2 RETURN c1,c2
```

Fuzzy linker is related to fuzzy tree and graph mining [21] where some patterns emerge from several graphs even if they do not occur exactly the same way everywhere regarding the structure.

Popular hotels may for instance be retrieved when they are chosen by many people. This is similar to the way popular people are detected if they are followed by a large number of people on social networks.

In Cypher, such queries are defined by using aggregators. For instance, the following query retrieves hotels visited by at least 2 customers:

Listing 9: Aggregation

```
MATCH (c:Customer)-[:VISIT]->(h:Hotel)
WITH h AS hotel, count(c) AS nbCustomer
WHERE nbCustomer > 1
RETURN hotel
```

Such crisp queries can be extended to consider fuzziness:

Listing 10: Aggregation

```
MATCH (c:Customer)-[:VISIT]->(h:Hotel)
WITH h AS hotel, count(c) AS nbCustomer
WHERE POPULAR(nbCustomer) > 1
RETURN hotel
```

The question then raised is now to implement them in the existing Neo4J engine.

3.2. Cypherf: Implementation Challenges

3.2.1. Architecture

There are several ways to implement fuzzy Cypher queries:

1. Creating an overlay language on top of the Cypher language that will produce well formatted queries expressed as Cypher statements to do fuzzy computations;
2. Extending the Cypher queries and using the existing low level API behind;
3. Extending the low level API with optimized functions only available to advanced users with development skills;
4. Combining the last two possibilities: using an extended Cypher query language over an enhanced low level API.

Every possibility is debated in this section and a synthesis is then provided.

3.2.2. Creating an Overlay Language

Concept

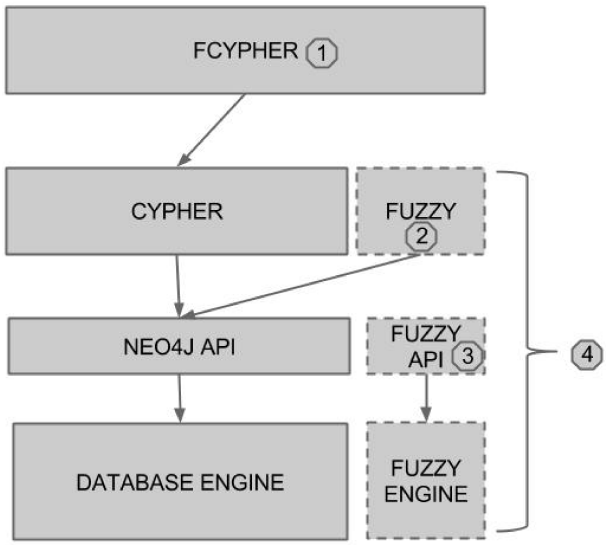


Figure 5: Implementation Ways

The concept is to create a new language dedicated to our framework. Such dedicated languages are also known as Domain Specific Languages (DSL). In this work, we consider a high-level fuzzy DSL that will be used to generate Cypher well-formed queries. The generated Cypher queries will be executed by the existing Neo4J engine.

A grammar must be defined for this external DSL which can rely on the existing Cypher syntax and only enhance it with new fuzzy features. The output of the generation process is pure Cypher code. In this scenario, Cypher is used as a low level language to achieve fuzzy queries.

Discussion

This solution is cheap and non-intrusive but has several serious drawbacks:

- Missing features: fuzzy queries (e.g., retrieving the data matching some fuzzy criterion) cannot be easily expressed by the current Cypher language (*e.g.,* Listing 4) without creating ad hoc queries;
- Performance issues: Cypher is neither designed for fuzzy queries nor for being used as an algorithmic language. All the fuzzy queries will produce

Cypher query codes that are not optimized for computations embedding fuzzy sets;

- Lack of user-friendliness: some queries cannot be executed directly against the Neo4J environment, it needs a three-step process: (i) write a fuzzy query, then compile it to get the Cypher query; (ii) use the Cypher generated queries on the Neo4J database; (iii) reconstruct a "fuzzy answer" from a regular Cypher query result

3.2.3. Extending the Cypher Queries

Concept

We propose to extend the Cypher language in order to add new features. Cypher offers various types of functions: scalar functions, collection functions, predicate functions, mathematical functions etc. To enhance this language with fuzzy features, we propose to add a new type of function: fuzzy functions. Fuzzy functions are used in the same way as other functions of Cypher (or SQL).

Cypher is an external DSL, which requires to be parsed. The query correctness must be checked and then it has to be executed. In Cypher the execution consists of retrieving the results from the query using the corresponding graph traversal.

The Neo4J's grammar provides the guidelines of how the language is supposed to be structured and what is and is not valid. In order to implement this solution, the Cypher grammar must be extended regarding the current grammar parser. Cypher uses the Scala Parser Combinator library (that will be covered in more details in Section 4) as parsing engine. Once the Cypher query is parsed, the code has to be bound on the current programmatic API to achieve the desired result.

Discussion

This work requires a deep understanding of the Neo4J engine and more knowledge of Java/Scala programming language (used to write the Neo4J engine and API) than the previous solution. The main advantage of this solution is to

offer an easy and user-friendly way to use the fuzzy feature. The disadvantages of this solution are:

- Performance issue. This solution should have better performance than the previous one. However, it is still built on the current Neo4J engine API that is not optimized for fuzzy queries (*e.g.* degree computing);
- Cost of maintenance. Until Neo4J accepts to merge this contribution to the Neo4J project, it will be required to upgrade every new version of Neo4J with these enhancements. Every new version of Neo4j can make some breaking changes such as API changes. If so, part of the fuzzy extensions will need to be rewritten.

3.2.4. Extending Low Level API

Concept

This scenario consists of enhancing the core database engine with a framework to efficiently handle the fuzzy queries and to extend the programming API that can be used by developers.

Discussion

This solution offers a high performance improvement but needs advanced Neo4J skills, possibly high maintenance costs, a poor user-friendliness experience (only developers can use it) and a costly development process.

3.2.5. Extending Cypher Over an Enhanced Low Level API

Concept

The last possibility is to combine the solutions from Sections 3.2.3 and 3.2.4: adding the features handling the fuzzy queries to the database engine extending the API while also extending the Cypher language.

Discussion

Although this solution is user-friendly and provides optimized performance, it has a heavy development cost (skills, tasks, etc.) and a high cost of maintenance.

Scenario	Price	Features	Optimized	User Friendliness
Overlay Language	++	−−	−−	−
Extending Cypher Queries	+	−	+	++
Extending Low Level API	+	++	−−	−
Cypher over an Enhanced Low Level API	−−	++	+++	++

Table 1: Discussion Sum-Up

Every scenario has pros and cons. Table 1 sums up the above discussion.

The best, but most costly, solution is still the last one: extending Cypher query language and build a low level API framework to enhance the Neo4J database engine with support of fuzzy queries as presented below.

3.3. First Implementation: Naive Implementation

Prototype

In [18], a prototype based on the extension of Cypher over an enhanced API has been introduced allowing fuzzy queries to be run as shown in Figure 6.

This first implementation is called "naive" because it deliberately rejects sophisticated architecture and is a standard "proof of concept" implementation of what has been specified in previous section.

Discussion

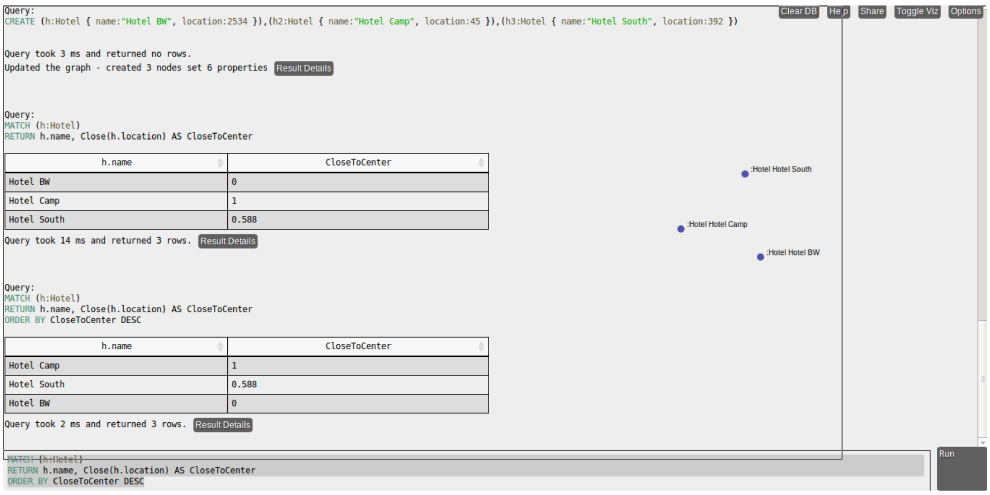


Figure 6: Protoype Developed

This implementation looks fairly user-friendly: a set of fuzzy functions to be used in Cypher queries are available to users. The main drawback of this implementation is that for each specific need a new function has to be developed.

For instance, as defined in section 3.1.1, to calculate the similarity between two nodes a function with two arguments (node1 and node2) has to be written. The calculation of the similarity between two hotels or two cars is based on different properties. This means that two different functions (SimilarityCars and SimilarityHotels) are needed. Each function makes assumption on some node properties (*e.g.* top speed, max power for a car).

The same problem occurs on fuzzy queries over properties: the μ_{cheap} fuzzy membership function is considered to calculate to which extent a property matches the “cheap" fuzzy term. The cheap function cannot be defined with the same μ_{cheap} membership function for a car or a hotel room: $\mu_{cheap_{car}}$ and $\mu_{cheap_{hotel}}$ have to be implemented in Scala, tested, compiled, and deployed in the Neo4J engine.

This is also a huge drawback when a user wants to define or refine a membership function: it is always a *code, build, run* cycle which requires the help of an developer or the possession of programming skills.

To conclude, this implementation is acceptable only for people who have few fuzzy cases to handle or have development skills to implement fuzzy functions by themselves.

This lack of genericity has highlighted the need for a new solution, described in the next section.

3.4. Cypherf: Extending Cypher to Fuzzy Queries

3.4.1. Overview

This new extension has been proposed and implemented to reduce the drawbacks listed in Section 3.3. The goal of this proposal is to provide an improved implementation on two axes: it must reduce the need for specific development and increase the capacity for the user to create his or her own membership function.

Experimentation and benchmarking are available in Section 5.

3.4.2. Cypher: an External Domain Specific Language

As said before, Cypher is an external Domain Specific Language for expressing queries on NoSQL graph databases. A Domain Specific Language (DSL) is a language tailored to a specific application domain [22].

DSLs are everywhere. Among the most popular are *SQL* for relational databases, *Make* for building softwares and *CSS* for styling description.

The main advantages of DSLs are that they are expressive, concise and designed at a high level of abstraction. Moreover, during the development cycle, working on DSL is more scalable and tends to produce a higher payoff. The main disadvantages are that the language design is hard and that DSLs lead to performance issues. There are two main categories of Domain Specific Languages:

- The first one is known as "internal" DSL. It uses the infrastructure of an existing programming language to build domain-specific semantics. It is implemented as a particular form of API in a host general purpose

language, often referred to as a “fluent" interface. The term *fluent* is used for an API that is primarily designed to be readable and to flow.

- The second category is named "external" DSL. This type of DSL is language agnostic, meaning that it is independent from a language syntax. It can thus be implemented in any language. One of the strengths of this category of DSL is its expressiveness, as it is not limited to a general/host programming language. The main drawback is that such languages have to be parsed to be manipulated. This requires the writing of a language parser or the use of a tool that uses a grammar to generate a parser for a specific language. For instance, AntLR [23] (ANother Tool for Language Recognition) can be considered for this task.

3.4.3. Cypherf: Adding a Fuzzy DSL to Cypher

The naive implementation extends the Cypher language with new functions: fuzzy functions.

This approach does not provide an effective way to express membership functions. To provide a more generic implementation, it is necessary to add another Domain Specific Language that allows the end-user to express fuzzy expressions.

The choice has been made to enhance Cypher with fewer functions but ones that allow the user to express fuzzy concerns with a "Fuzzy DSL". This new extension of Cypher is called Cypherf, standing for Cypherfuzzy.

3.4.4. Features

Cypherf allows the end-user to use fuzzy features such as defining membership function with the use of a fuzzy DSL, using t-norms and t-conorms functions as defined in [24] and using fuzzy linguistic variables.

The available functions are listed below (a formal grammar definition is available in Section 4.2.3):

- Fuzzy(μ_f, value): returns the degree of membership of the μ_f function

- μ_f describes the membership function with the Fuzzy DSL
- value is expressed as a Double

- FuzzyLT(fuzzyVariable, value): returns a collection that contains for every fuzzy term of the fuzzy linguistic variable two properties: the name of the term and the degree of compatibility of a "value" with the term. For instance, for a value X and a fuzzy linguistic variable $Age = (Age, [0, 130], \{young, middleaged, old\}, \{\mu_{young}, \mu_{middle}, \mu_{old}\})$ the result will be:

```
{
    {name:"young",degree:μ_young(x)},
    {name:"middle",degree:μ_middle(x)},
    {name:"old", degree:μ_old(x)}
}
```

- fuzzyVariable: is expressed as a String that defines the fuzzy variable. This definition is composed of the name of the fuzzy variable and a set of fuzzy-terms. Every fuzzy term is defined by its name and its membership function
- value is expressed as a Double

- TNorms(tnormName, expression1, expression2): applies a TNorm of name "tnormName" on expression1 and expression2

- TCoNorms(type, expression1, expression2): same as TNorms but this time for t-conorms.

3.4.5. Fuzzy DSL

The fuzzy DSL is used to define membership functions and fuzzy linguistic variables.

It is processed by an underlying fuzzy framework called "Fuzzy4S" that implements the fuzzy logic. Fuzzy4S has been developed explicitly as part of this new implementation but can be used separately in other projects. The

grammar of the fuzzy DSL is explained in the section 4.2.3 after the presentation of the Fuzzy4S framework and the introduction of some underlying concepts.

In this section, some contribution about defining and using fuzzy queries in a graph database has been proposed. Several implementation scenarios have been discussed and two implementations confronted. Implementations are based on an extension of the declarative graph database query language Cypher and the development of a low level engine that offers fuzzy features. The last implementation is more generic and user-friendly because it does not need programming skills to be used.

The next section presents the underlying fuzzy framework that has been developed for Cypherf.

4. Fuzzy4S: A Fuzzy Logic Framework for Scala

In this contribution, approximate queries are considered in the way they are defined using fuzzy set theory.

4.1. Overview

In this paper, we introduce Fuzzy4S which is a fuzzy logic framework written in Scala. Proposed as a library, it includes membership functions, t-norms and t-conorms [10]. Upon this library, an open Domain Specific Language (DSL) has been built to define approximate queries at an abstract level. It relies on the IEC 61131 standard (IEC61131-7) that had been written for fuzzy control programming and on jFuzzyLogic (a java fuzzy logic library) [25, 26].

The need for such contributions in the fuzzy set framework has been highlighted in [27].

The rest of this section introduces our Domain Specific Language and the underlying features.

4.2. Fuzzy4S Grammar

Fuzzy4S can be manipulated directly (by using a Domain Specific Language based on a grammar that will be presented later in this section) or programmatically (by referencing objects and classes of the framework written in Scala). As written in [28], *Scala is implemented as a translation to standard Java bytecode*, which implies that the Scala code can be used in Java.

In the case of the Fuzzy4S DSL, the choice has been made to rely on external DSL which are more expressive than internal DSL. We have thus written a parser based on the concept of parsing expression grammar (PEG) which can be viewed as an extension of EBNF [29] and that is introduced below.

4.2.1. Fuzzy4S: Parsing Expression Grammar (PEG)

A Parsing Expression Grammar (PEG) is a recognition-based grammar formalism. The recognition-based systems have been developed in the 1970s. For decades, the Chomsky's generative system of grammars, particularly context-free grammars (CFGs) and regular expressions (REs), has been used to express the syntax of programming languages and protocols [30].

As said in [31], *the power of generative grammars to express ambiguity is crucial to their original purpose of modelling natural languages, but this very power makes it unnecessarily difficult both to express and to parse machine-oriented languages using CFGs.*

A PEG provides an alternative for describing machine oriented syntax, which solves the ambiguity problem by not introducing ambiguity in the first place. Where CFGs express non-deterministic choice between alternatives, PEGs instead use prioritized choice by trying the alternatives in their order and unconditionally consuming the first successful match.

Figure 7 shows the notation of common operators available in PEG and the equivalent notation in Scala.

PEG operators are used in Fuzzy4S to parse the external DSL with a combination of several parsers. This technique is called combinatory parsing.

Description	PEG notation	Scala notation
Literal string	' '	" "
Literal string	" "	""" """
Character class	[ ]	"[ ]".r
Any character	.	".".r
Grouping	(*e*)	(e)
Optional	e?	(e?) or opt(e)
Zero-or-more	e*	(e*) or rep(e)
One-or-more	e+	(e+) or rep1(e)
And-predicate	&e	guard(e)
Not-predicate	!e	not(e)
Sequence	$e_1\, e_2$	e1 ~ e2
Prioritized Choice	e_1 / e_2	e1 | e2

Figure 7: PEG operators [31] [32]

4.2.2. Combinatory Parsing

In functional programming, a common approach to parse domain-specific languages is to model parsers as functions and to define higher-order functions (also called combinators) that implement grammar constructions such as sequencing, choice and repetition. As explained in [33], the basic idea dates back to 1970s [34] and has become popular since 1980s in a variety of functional programming languages [35, 36, 37].

4.2.3. Fuzzy4S Parsing Expression Grammar

The parsing expression grammar for the DSL used in Fuzzy4S is defined in Listing 11.

Listing 11: Fuzzy4S Parsing Expression Grammar

```
1
2
3   def number: Parser[Double] = """-?\d+(\.\d*)?"""
4   def point: Parser[Point] = "(" ~> number ~ "," ~ number <~ ")"
5   def functionName: Parser[String] = """trian|trape|gauss|gbell|sigm↩
        """"
6   def parameters = rep1(point) | rep1(number)
7   def memberfunction = functionName.? ~ parameters
```

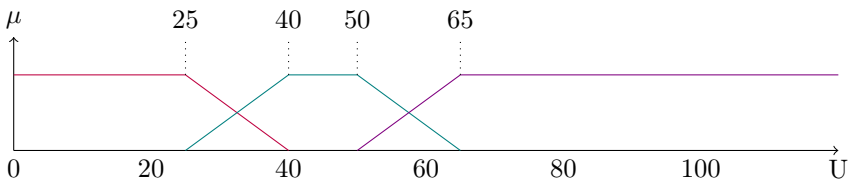


Figure 8: Result of listing 12

```
    def termName = """[a-zA-Z]\w*"""
    def term =  "TERM" ~> termName ~ ":=" ~ memberfunction <~ ";"
    def variableName = """[a-zA-Z]\w*"""
    def fuzzyVariable = "FUZZIFY" ~> variableName ~ rep1(term) <~ "↩
        END_FUZZIFY"
```

Listing 12: Fuzzy variable example: fuzzy linguistic definition of Age

```
FUZZIFY age
        TERM young:=  (25,1) (40,0);
        TERM middle:= trape 25 40 50 65;
        TERM old:= (50,0) (65,1);
END_FUZZIFY
```

As defined in Listing 11, a point can be expressed as a pair of two numbers, the first one defining the x coordinate and the second one the y coordinate. A membership function has an optional function name and some parameters (point or number). A Term defines a fuzzy set, it is composed of a name and a membership function. A fuzzy linguistic variable (named fuzzyVariable in Listing 11) is composed of a name, for instance *age*, and all the terms that compose this fuzzy linguistic variable (*young*, *old*, ...) such as defined in Listing 12 and represented by Figure 8.

Fuzzy linguistic variable definition can be stored in files and can be loaded in Fuzzy4S by using the file path. This choice has been made in order to provide a simple way for users to create their own library to define their specific linguistic terms.

One of our potential future project's feature will be to provide a web ap-

plication where users can publish their definition of fuzzy linguistic variable to compose an open source repository. Users will be able to load definition from this site by using HTTP URIs.

4.3. Epitomization

We claim that fuzzification is a challenge for graph databases. Our contribution is to propose an extension of the Cypher language, one of the most widely adopted graph database query language, to handle fuzzy queries.

Our first implementation attempt required to create a new function for each usecase and context (e.g., Cheap, Far, Cold, etc.). Each function implementation has to be invocated by the use of a *FunctionInvocation* (in Cypher's EBNF grammar).

We think that this approach was lacking of genericity. We thus decided to offer a better support of fuzzy features to the end users through the definition of a domain specific language (DSL) called Fuzzy4S that provides the user with the ability to define his own fuzzy linguistic vocabulary and terms with associated functional membership functions and to chain them in their Cypherf queries.

Upon that, our choice of extending Cypher (instead of creating a separated language) offers the user more flexibility by the possibility to use functions (e.g. contains()) and operations (e.g. head()) of the underlying system.

The next section introduces some examples that show the power of our contribution.

5. Examples and Experimentation

This section illustrates our contributions with examples and shows the efficiency of our implementation with experimental load tests over benchmarks.

5.1. Using Fuzzy4S

This section is divided into two parts. The first is a showcase of some classical examples of what can be done with Fuzzy4S. The second introduces a web application that offers a web console to try out the Fuzzy4S framework.

Membership function	Scala code	DSL
PieceWise	PieceWiseMembershipFunction (List(ValuePoint(0,1), ValuePoint(2,1), ValuePoint(4,0)))	(0,1) (2,1) (4,0)
Triangular	TriangularMembershipFunction(0,2.5,5)	trian 0 2.5 5
Trapezoidal	TrapezoidalMembershipFunction(0,2,3,4)	trape 0 2 3 4
Sigmoidal	SigmoidalMembershipFunction(-4, 3)	sigm -4 3
Singleton	SingletonMembershipFunction(SingletonPoint(2))	2
...	...	...

Figure 9: Examples of membership functions using Scala4S

5.1.1. Examples

In Figure 9 some examples of using membership functions of Fuzzy4S are provided both in Scala code and with the external DSL.

5.1.2. Try Fuzzy4S: a Web Console

A web application is proposed to test the Fuzzy4S functionality. This application is a web console that offers the opportunity to define and evaluate some membership functions with the Fuzzy4S DSL as shown in Figure 10.

5.2. Using Cypherf

This section uses the example of searching for the best hotel for a trip to Paris. Two criteria will be explored: the price of the hotel for one night and the distance to the center of the city.

This section contains examples which can be seen as a step-by-step tutorial:

- example 1 and example 2 introduce basic Cypher query concepts required for the rest of this section;
- example 3 and example 4 introduce the use of membership functions with Cypherf;
- example 5, example 6 and example 7 illustrate how to use linguistic variables;

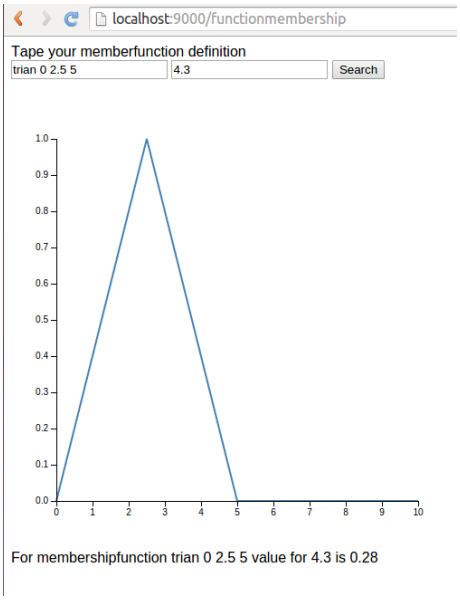


Figure 10: Fuzzy4S web console

- example 8 and example 9 use t-norms and t-conorms.

5.2.1. Data modelization and preliminary examples

Hotel and city center can be modelized as shown in Figure 11.

Using modelization from Figure 11, finding hotels close to the city center of the town requires calculating the distance from the hotel to the center of the city. The Euclidean distance between two points of the plane with Cartesian coordinates (x_1, y_1) and (x_2, y_2) is

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

Example 1. *Calculate the Euclidean distance in a graph database using Cypher*

Listing 13: Calculate the euclidean distance between a hotel and the city center

```
match (c:Center),(h:Hotel)
with h.name as hotel, sqrt((c.posX - h.posX) * (c.posX - h.posX)↩
    + (c.posY - h.posY) * (c.posY - h.posY)) as distance
```

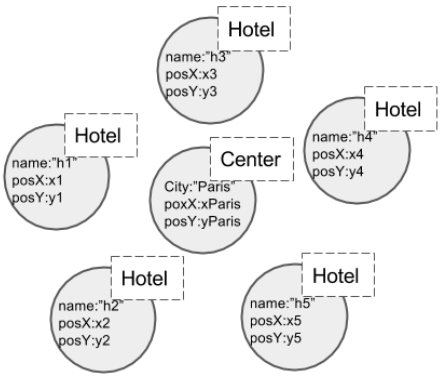


Figure 11: a possible data modelization

```
3  return hotel, distance
4  order by distance
```

Listing 13 shows how to calculate the Euclidean distance. The first line of Listing 13 specifies which node will be searched in the request, in this example all the nodes with label "Hotel" or label "Center". The WITH instruction manipulates the output before passing it to the rest of the query. It is a sort-of "pipe" command. In this example it takes hotels and center nodes and returns the hotel name to the rest of the query as a property called "hotel" and the distance expressed from the euclidean distance between the node and the center of the city.

In Example 1 the calculation of the euclidean distance must be done for each query execution. A better approach could be to store distance information into a relationship between the hotel and the city center.

The same technique can be considered for the result of the execution of a membership function like "close". It can also be stored in the same relationship.

Example 2. *Store processed information in a relationship*

Listing 14 shows how to store the result of the euclidean distance calculation in a relationship between hotel and city center.

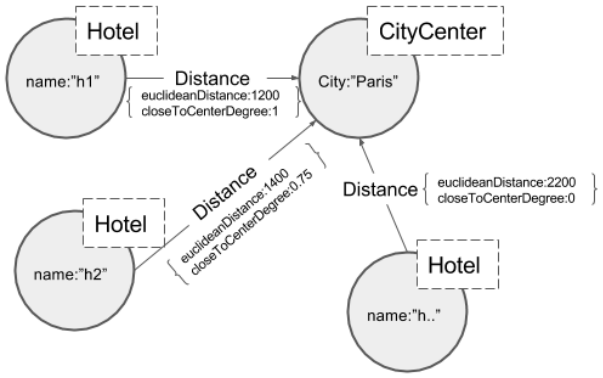


Figure 12: a data modelization with relationships

At the end of the query execution, the graph model looks like: (h:Hotel)-[r:Distance distance:VALUE]->(c:Center) as represented in Figure 12.

Listing 14: Adding a relationship with euclidean distance

```
MATCH (c:Center),(h:Hotel)
WITH h, c, sqrt((c.posX - h.posX) * (c.posX - h.posX) + (c.posY - h.posY) * (c.posY - h.posY)) as distance
WITH h, c, distance, fuzzy("(1200,1) (2000,0)", distance) AS closeToCenterDegree
CREATE (h) -[r:Distance {euclideanDistance:distance, closeToCenter:closeToCenterDegree}]->(c)
return h,r,c
```

In order to focus on the fuzzy expression of Cypherf language, the model has been transformed. The hotels no longer have *posX* and *posY* properties, only a distance property that expresses the distance between the hotel and the center of Paris. The resulting data set is shown in Listing 15 and represented in Figure 13

Listing 15: Dataset

```
CREATE (h1:Hotel { name:"h1", price:60, distance:3200})
CREATE (h2:Hotel { name:"h2", price:70, distance:2400})
```

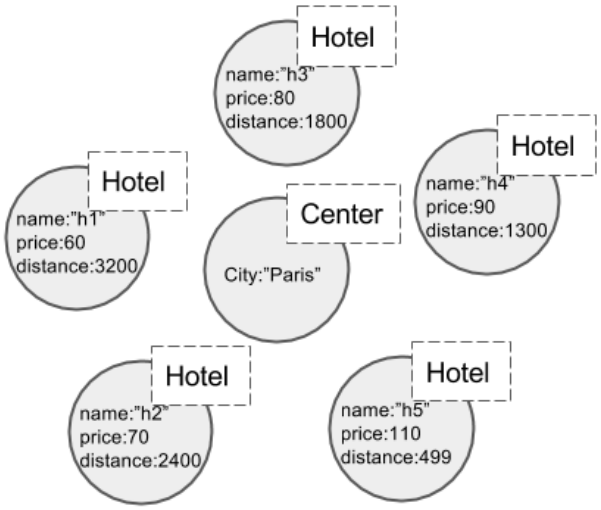


Figure 13: Figure of listing 15

```
CREATE (h3:Hotel { name:"h3", price:80, distance:1800})
CREATE (h4:Hotel { name:"h4", price:90, distance:1300})
CREATE (h5:Hotel { name:"h5", price:110, distance:499})
CREATE (c:Center { city:"Paris"})
```

5.2.2. Using a Membership Function

In this section, the goal is to retrieve the hotels close to the city center. To do so a piecewise membership function, illustrated in Figure 14, is defined as:

$$f(x) = \begin{cases} 1 & 0 \leq x \leq 1200 \\ \frac{-x}{800} + \frac{5}{4} & 1200 < x < 2000 \\ 0 & x >= 2000 \end{cases}$$

Example 3. *Using a closeToCenter piecewise membership function in Cypherf*

Listing 16 shows the use of the "fuzzy" Cypherf function that allows the user to use the Fuzzy DSL to describe a membership function. The result is the degree of membership of the hotel's distance to the defined membership function.

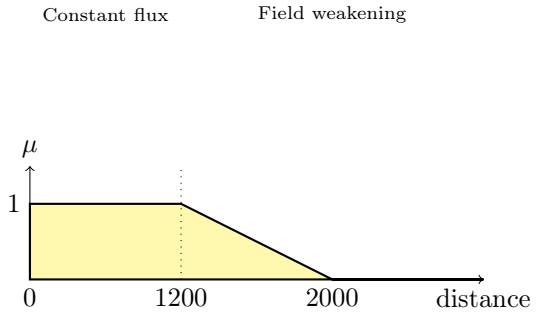


Figure 14: Distance membership function

```
$ match (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance return h.name, distance order by distance desc
```

h.name	distance
h5	1
h4	0.875
h3	0.25
h1	0
h2	0

Returned 5 rows in 64 ms.

Figure 15: Result of querying distance to retrieve hotel

Listing 16: Querying distance to retrieve hotel by closeToCenter degree

```
MATCH (h:Hotel)
WITH h, fuzzy("(1200,1) (2000,0)", h.distance) AS distance
RETURN h.name, distance
ORDER BY distance DESC
```

Figure 15 shows the result of listing 16.

Example 4. *Using a threshold on the degree of membership to closeToCenter*

Listing 17 retrieves the first ten results with a degree of 0.8 *at least. Figure 16 shows the results.*

Listing 17: Retrieving hotels close to city center with a threshold on degree of membership to closeToCenter, limit to the top three results

```
MATCH (h:Hotel)
WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance
WHERE distance > 0.8
```

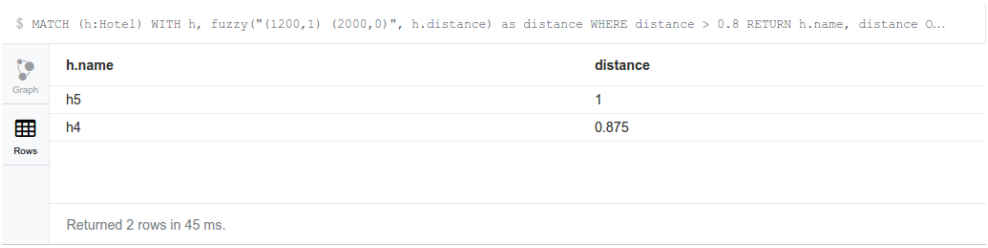


Figure 16: Result of querying distance with threshold

```
RETURN h.name, distance
ORDER BY distance DESC
```

We have seen in this section how to use a membership function inside Cypherf query language. The next section shows how to use linguistic variables in Cypherf.

5.2.3. Using a Linguistic Variable

Let a linguistic variable price be defined with 3 terms: cheap, midPrice, expensive.

To allow users to retrieve the degree of membership associated to each term of a fuzzy linguistic variable, all membership functions must be applied. It is thus necessary to provide a "price.fl" (fl stands for FuzzyLogic) file that contains the variable definition as shown in Listing 18.

Listing 18: definition of price linguistic variable

```
FUZZIFY price
        TERM cheap:= (70,1) (85,0);
        TERM midPrice:= trape 75 85 90 105;
        TERM expensive:= (95,0) (110,1);
END_FUZZIFY
```

Example 5. *Using Cypherf with a linguistic variable*

If the linguistic variable definition is saved in the /tmp/price.fl location then Cypherf query can be used to build a query as in Listing 19.

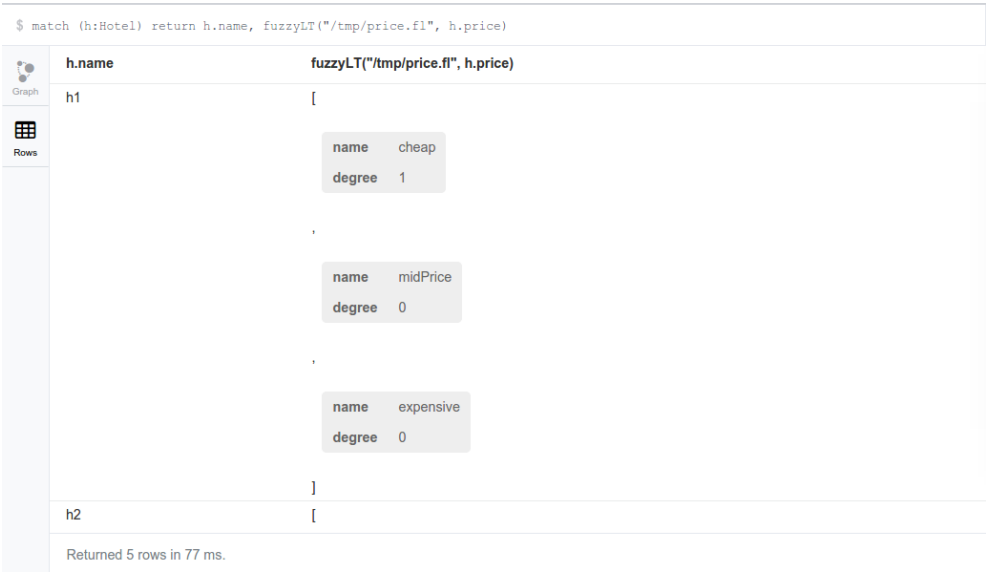


Figure 17: Result of querying linguistic variable price

Listing 19: Using Cypherf with a linguistic variable

```
match (h:Hotel)
return h.name, fuzzyLT("/tmp/price.fl", h.price)
```

As shown in Figure 17, the query returns a collection of terms composed of a name and a degree. This collection is sorted by decreasing degree and (if two terms have the same degree) by term declaration order in the linguistic variable.

Example 6. *Retrieving the most appropriate term of a linguistic variable*

Listing 20 shows Cypher query to retrieve the **top** *term of this linguistic variable for each hotel, meaning the most appropriate term of the price linguistic variable for each hotel.*

Listing 20: Retrieving the most appropriate term of a linguistic variable

```
MATCH (h:Hotel)
WITH h, h.price as price
RETURN h.name, head(fuzzyLT("/tmp/price.fl", price))
```

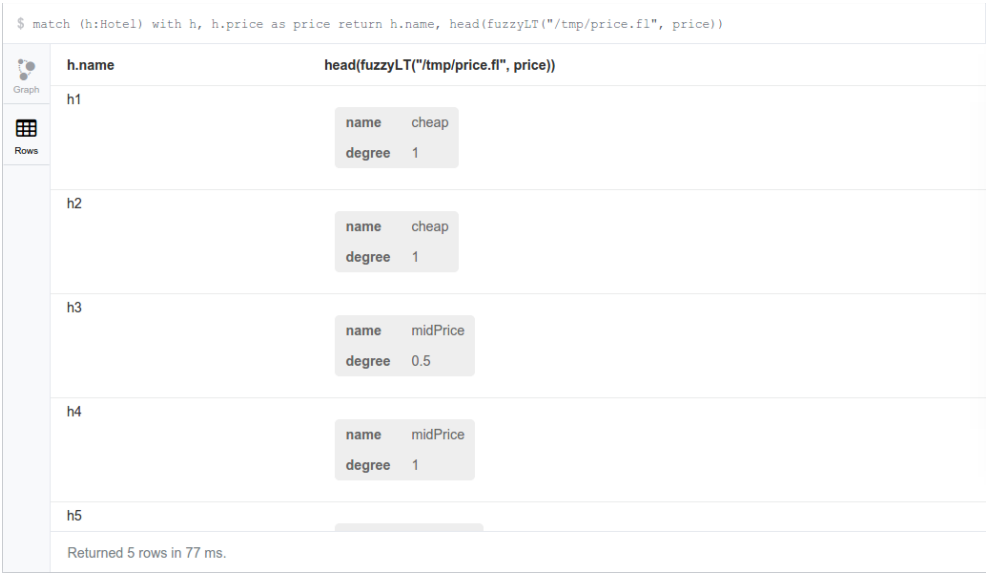


Figure 18: Result of querying the top term of linguistic variable price

Line 3 of Listing 20 shows the fuzzyLT function returning the head of the collection, which is to say the element with the highest degree and declared first in the linguistic terms. Figure 18 shows that each returned element has two properties: a name and a degree.

Listing 21 illustrates the name of the hotels and the top price term name. Results are available in Figure 19.

Listing 21: Display the best price term name

```
1 MATCH (h:Hotel)
2 WITH h, h.price AS price
3 RETURN h.name, head(fuzzyLT("/tmp/price.fl", price)).name
```

Example 7. *Storing fuzzy information in nodes*

As seen in Listing 14, information processed from the fuzzy functions can be stored back in the graph. For instance, for every hotel, the top-price name can be stored as a property of the hotel node. Listing 22 illustrates how to store the

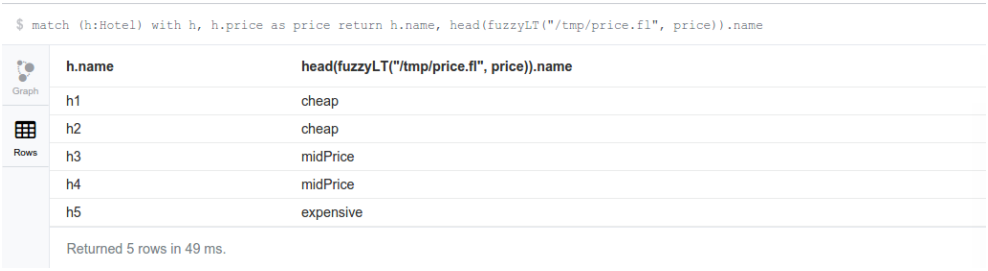


Figure 19: Result of querying the name of the top term of linguistic variable price

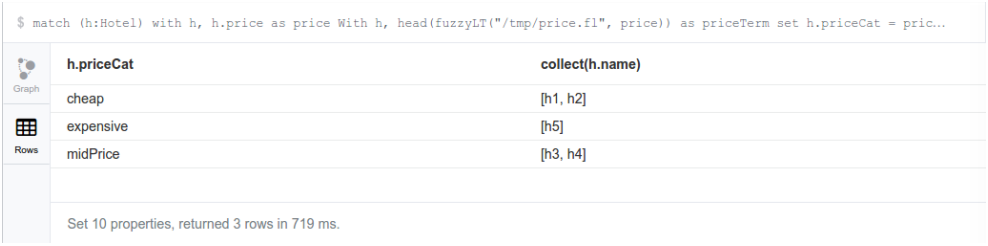


Figure 20: Result of storing category in Hotel node and grouping by category

most appropriate term in hotel node and shows that data can be grouped based on fuzzy information. Figure 20 shows the result of Listing 22.

Listing 22: Store back most appropriate term in a node, group by Category

```
MATCH (h:Hotel)
WITH h, h.price AS price
WITH  h, head(fuzzyLT("/tmp/price.fl", price)) AS priceTerm
SET h.priceCat = priceTerm.name, h.priceDegree = priceTerm.degree
RETURN h.priceCat, collect(h.name)
```

5.2.4. Combining Fuzzy Terms

In this example section, the goal is to find the hotels close to the city center and have low price. To do so, two linguistic variables are used: the price and the distance of the hotel to the city center. The price definition remains the same as in listing 18 and the fuzzy linguistic variable **distance** is defined as in Listing 23.

Listing 23: Distance definition

```
FUZZIFY distance
        TERM close:= (1200,1) (2000,0);
        TERM medium:= trape 1200 2000 2500 3000;
        TERM far:= (2500,0) (3000,1);
END_FUZZIFY
```

Example 8. *Using t-norms with two fuzzy linguistic variables*

Listing 24 shows the Cypherf query to achieve this goal and Figure 21 shows the results when using the min t-norm.

Listing 24: Using TNorm

```
Match (h:Hotel)
WITH h as hotel, h.price as hotelPrice, h.distance as ↩
    hotelDistance
WITH hotel, FuzzyLT("/tmp/distance.fl", hotelDistance) as ↩
    distanceTerms, FuzzyLT("/tmp/price.fl", hotelPrice) as ↩
    priceTerms
UNWIND distanceTerms as distance
UNWIND priceTerms as price
return hotel.name, distance.name, price.name, TNorm("Min", ↩
    distance.degree, price.degree) as minR order by TNorm("Min", ↩
    distance.degree, price.degree) desc
```

Example 9. *Final example: finding the best hotels*

Listing 25: Using TNorm and where clause

```
MATCH (h:Hotel)
WITH h AS hotel, h.price AS hotelPrice, h.distance AS ↩
    hotelDistance
WITH hotel, FuzzyLT("/tmp/distance.fl", hotelDistance) AS ↩
    distanceTerms, FuzzyLT("/tmp/price.fl", hotelPrice) AS ↩
    priceTerms
UNWIND distanceTerms AS distance
UNWIND priceTerms AS price
With hotel, distance, price
WHERE price.name = "cheap" and distance.name = "close"
```

hotel.name	distance.name	price.name	minR
h1	far	cheap	1
h2	medium	cheap	1
h5	close	expensive	1
h4	close	midPrice	0.875
h3	medium	midPrice	0.5
h3	medium	cheap	0.33333333333333337
h3	close	midPrice	0.25
h3	close	cheap	0.25
h4	medium	midPrice	0.125
h1	far	midPrice	0
h1	far	expensive	0
h1	close	cheap	0
h1	close	midPrice	0
h1	close	expensive	0
h1	medium	cheap	0
h1	medium	midPrice	0

Returned 45 rows in 81 ms.

Figure 21: Using t-norm

hotel.name	distance.name	price.name	minR
h3	close	cheap	0.25
h1	close	cheap	0
h2	close	cheap	0
h4	close	cheap	0
h5	close	cheap	0

Returned 5 rows in 78 ms.

Figure 22: Result of using TNorm with where clause

```
RETURN hotel.name, distance.name, price.name, TNorm("Min", distance.degree, price.degree) AS minR ORDER BY TNorm("Min", distance.degree, price.degree) DESC
```

Figure 22 shows that only one hotel, the cheapest one, could fit with our request. Nevertheless, sometimes the least expensive option is not the best match. We thus change the price criterion from "cheap" to "midPrice" as shown in Listing 26.

Listing 26: Requesting with price=midPrice and distance=Close

```
MATCH (h:Hotel)
```

Figure 23: Result of query with price=midPrice and distance=Close

```
WITH h AS hotel, h.price AS hotelPrice, h.distance AS hotelDistance
WITH hotel, FuzzyLT("/tmp/distance.fl", hotelDistance) AS distanceTerms, FuzzyLT("/tmp/price.fl", hotelPrice) AS priceTerms
UNWIND distanceTerms AS distance
UNWIND priceTerms AS price
WITH hotel, distance, price
WHERE price.name = "midPrice" and distance.name = "close"
RETURN hotel.name, distance.name, price.name, TNorm("Min", distance.degree, price.degree) AS minR ORDER BY TNorm("Min", distance.degree, price.degree) DESC
```

Result of listing 26 is shown in Figure 23.

5.3. Load Testing

5.3.1. Experimental Design

A test protocol has been defined to minimize the side effects (of both the dynamic compilation as well as the JVM memory management) on performance measurement based on works made in [32].

There are some mechanisms in the JVM which are transparent to the programmer, like automatic memory management, dynamic compilation, adaptive optimization and so on [38, 39]. These mechanisms are triggered implicitly and can have an enormous impact on measurements. For example:

- Just in Time (JIT) compilation: during the program the same code might exhibit very different performance characteristics. The HotSpot compiler could potentially compile any part of the code at any point during the

runtime. This can happen in the middle of running a benchmark, yielding an inaccurate running time measurement;

- Classloading: JVM has global program information available (unlike a typical compiler). One method may be optimized based on the information in some unrelated method;

- Automatic memory management: the execution of running code can lead to the triggering of garbage collection cycle which changes the observed performance of the code being tested.

To generate better performance measurement, a test protocol has been defined and applied on every test:

- A separate JVM executor is created. Having a separate JVM ensures that only the bare essentials are needed to run the tests. It prevents the JIT compiler from applying some optimizations to the code.

- This executor has a default heap size set to 2GB

- To ensure that the JIT compiler optimized the code appropriately, the tests code monitors the running time during the warmup to dynamically detect if the running time has become stable

- A warmup is done before every test using statistical data analysis such as confidence interval as described in [40]

```
[info] Running test set for Fuzzy4S.membershipfunction, curve Test-0
[info] Starting warmup.
[info] 0. warmup run running time: 60 (covNoGC: NaN, covGC: NaN)
...
[info] 20. warmup run running time: 30 (covNoGC: 0.060, covGC: 0.354)
[info] Steady-state detected.
[info] Ending warmup.
```

- 36 repetitions of a measurement[5] for every input size before moving on to benchmarking wider ranges
- a Min aggregator is used to retrieve the minimum running time of all the benchmarks run for each size

5.3.2. Fuzzy4S Measurement

The goal of this section is to determine if Fuzzy4S performs as well as a state-of-the-art fuzzy engine written on the JVM. To do so, Fuzzy4S is compared to JFuzzyLogic [25] a fuzzy logic framework written in Java.

The performance scenario is described below.

- Use a range generator that generates an inclusive range of integer values. For instance for an x-axis of 10, it generates values 0,1,..,10.
- Use a triangular membership function.
- Calculate the running time of JFuzzyLogic and Fuzzy4S frameworks to parse the membership function and evaluate each value of x for every range

The result of the benchmark in Figure 24 shows, for this request, better performance for Fuzzy4S than JFuzzyLogic, meaning that Fuzzy4S is an efficient fuzzy engine implementation.

5.3.3. Cypherf Measurement

This section aims at evaluating the additional runtime costs (overhead) of Cypherf. This consists of discussing whether or not the extension of Cypher and the use of the Fuzzy4S engine handling fuzzy queries are acceptable. Listing 27 is a pure Cypher query that retrieves hotels' name whereas query of Listing 28 returns for every hotel its name and the most appropriate price term.

[5] Using 36 measurements increases the probability to have at least 30 measures after removing outliers due to external events (such as garbage collection), which allows us to tighten the confidence interval.

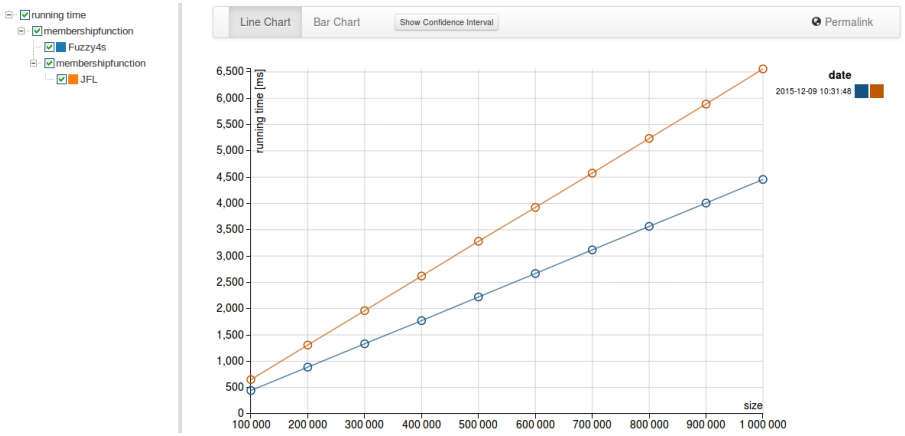


Figure 24: Benchmark comparison on 1 million nodes between Fuzzy4S and JFuzzyLogic frameworks

To better highlight performance measurement of fuzzy runtime overhead, a LIMIT 10 clause has been added. Indeed, the deserialization of query results is an important part of the overall time of each query.

Listing 27: Pure Cypher query in Neo4j

```
MATCH (h:Hotel) RETURN h.name LIMIT 10
```

Listing 28: Cypherf query on Neo4j-Fuzzy

```
MATCH (h:Hotel)
WITH h, h.price as price
RETURN h.name, head(fuzzyLT("/opt/research/data/price.fl", price))
LIMIT 10
```

Figure 25 shows that there is no major difference between Cypher and Cypherf execution. The additional cost of using fuzzy statements with Cypherf is thus compatible with the use of NoSQL graph databases.

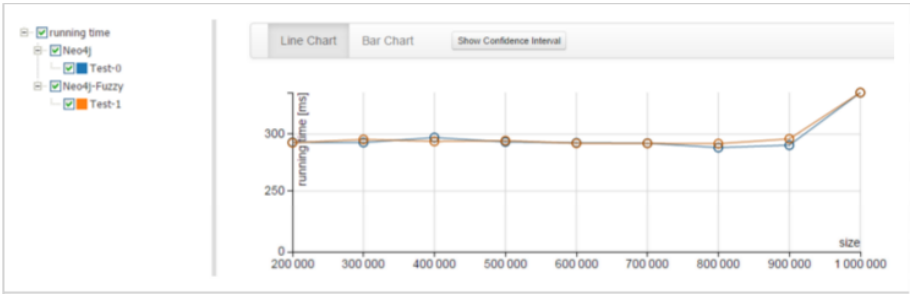


Figure 25: Benchmark comparison on 1 million nodes between a Cypher and a Cypherf query

6. Conclusion

Graph databases are becoming more and more used by both scientists and practitioners as the available engines are more and more powerful and scalable. They are used in many application domains, for the industry (e.g., logistics), ecommerce (e.g., ebay delivery service) or social networks (e.g., LinkedIn). Their ability to compute complex queries is a key feature. However, there is not yet any native possibility to compute fuzzy queries.

In this article, we thus introduce the necessary frameworks to support approximate queries in NoSQL graph databases. Fuzzy4S provides a framework via a Domain Specific Language (DSL) to define approximate linguistic variables. This DSL can then be used for querying NoSQL graph databases with Cypherf which extends the existing declarative language to manage approximate queries. Our propositions have been tested, we also have shown that the overhead is low assessing the power of the primitives that have been introduced.

This work opens up many research topics and possibilities. Regarding Big Data, many real-world applications and uses could be challenging for our approach. Fuzzy information may indeed be embedded within the data, especially in the (*key*; *value*) properties, as for instance Hotel $h1$ could be located at *around 800 meters* from the city center. It is also the case in social networks for all numerical properties on people and organizations (age, salary, size, revenue, etc.) on which various fuzzy variables may be defined depending on the context

and/or application.

In particular, further work may focus on the definition of complex operations in such a context of schema-less databases, as for instance the definition of the *SimilarTo* function on nodes that have to be compared while they may have disjoint sets of properties.

Moreover, Fuzzy4S and Cypherf may be used to retrieve fuzzy linguistic summaries from large NoSQL databases. Such approaches require to apply complex queries and will better highlight the power of graph databases (e.g., paths).

References

[1] A. laszlo Barabasi, Linked: How Everything Is Connected to Everything Else and What It Means for Business, Science, and Everyday Life, Basic Books, 2014.

[2] A. Castelltort, A. Laurent, Rogue behavior detection in nosql graph databases, J. Innovation in Digital Ecosystems 3 (2) (2016) 70–82. doi:10.1016/j.jides.2016.10.004.
URL http://dx.doi.org/10.1016/j.jides.2016.10.004

[3] R. Angles, C. Gutierrez, Survey of graph database models, ACM Comput. Surv. 40 (1) (2008) 1–39.

[4] T. T. A. Board, Technology radar (2013).

[5] M. Odersky, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, M. Zenger, et al., An overview of the scala programming language (second edition), Tech. rep. (2006).

[6] I. Robinson, J. Webber, E. Eifrem, Graph Databases, O'Reilly, 2013.

[7] R. E. Pattis, Teaching EBNF first in CS 1, in: R. Beck, D. Goelman (Eds.), Proceedings of the 25th SIGCSE Technical Symposium on Computer Sci-

ence Education, 1994, Phoenix, Arizona, USA, March 10-12, 1994, ACM, 1994, pp. 300–303. doi:10.1145/191029.191155.
URL http://doi.acm.org/10.1145/191029.191155

[8] D. E. Knuth, backus normal form vs. backus naur form, Commun. ACM 7 (12) (1964) 735–736. doi:10.1145/355588.365140.
URL http://doi.acm.org/10.1145/355588.365140

[9] P. Smets, Imperfect information: Imprecision and uncertainty, in: A. Motro, P. Smets (Eds.), Uncertainty Management in Information Systems, Springer US, 1997, pp. 225–254.

[10] L. Zadeh, Fuzzy sets, Information and Control 8 (3) (1965) 338 – 353.

[11] P. Bosc, O. Pivert, SQLf: a relational database language for fuzzy querying, Fuzzy Systems, IEEE Transactions on 3 (1) (1995) 1–17.

[12] Y. Takahashi, A fuzzy query language for relational databases, IEEE Transactions on Systems, Man, and Cybernetics 21 (6) (1991) 1576–1579.

[13] D. Dubois, H. Prade, Using fuzzy sets in flexible querying: Why and how?, in: T. Andreasen, H. Christiansen, H. Larsen (Eds.), Flexible Query Answering Systems, Kluwer Academic Publishers, Boston, 1997, pp. 45–60.

[14] S. Zadrozny, J. Kacprzyk, Implementing fuzzy querying via the internet/www: Java applets, activex controls and cookies., in: T. Andreasen, H. Christiansen, H. L. Larsen (Eds.), FQAS, Vol. 1495 of Lecture Notes in Computer Science, Springer, 1998, pp. 382–392.

[15] J. Galindo, J. M. Medina, O. Pons, J. C. Cubero, A server for fuzzy sql queries., in: T. Andreasen, H. Christiansen, H. L. Larsen (Eds.), FQAS, Vol. 1495 of Lecture Notes in Computer Science, Springer, 1998, pp. 164–174.

[16] J. Z. Pan, G. B. Stamou, G. Stoilos, S. Taylor, E. Thomas, Scalable querying services over fuzzy ontologies, in: J. Huai, R. Chen, H.-W. Hon, Y. Liu,

W.-Y. Ma, A. Tomkins, X. Z. 0001 (Eds.), WWW, ACM, 2008, pp. 575–584.

[17] J. Cheng, Z. M. Ma, L. Yan, f-sparql: A flexible extension of sparql, in: P. G. Bringas, A. Hameurlain, G. Quirchmayr (Eds.), DEXA (1), Vol. 6261 of Lecture Notes in Computer Science, Springer, 2010, pp. 487–494.

[18] A. Castelltort, A. Laurent, Fuzzy queries over nosql graph databases: Perspectives for extending the cypher language, in: International Conference on Processing and Management of Uncertainty in Knowledge-Based Systems, Springer, 2014.

[19] M.-J. Lesot, M. Rifqi, H. Benhadda, Similarity measures for binary and numerical data: a survey, International Journal of Knowledge Engineering and Soft Data Paradigms 1 (1) (2008) 63–84. doi:10.1504/ijkesdp.2009.021985.
URL https://hal.inria.fr/hal-01072737

[20] E. Panzeri, G. Pasi, A flexible extension of xquery full-text, in: R. Basili, F. Sebastiani, G. Semeraro (Eds.), Proceedings of the 4th Italian Information Retrieval Workshop, Pisa, Italy, January 16-17, 2013, Vol. 964 of CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 29–32.
URL http://ceur-ws.org/Vol-964/paper4.pdf

[21] F. D. R. López, A. Laurent, P. Poncelet, M. Teisseire, FTMnodes: Fuzzy tree mining based on partial inclusion, Fuzzy Sets and Systems 160 (15) (2009) 2224–2240.

[22] M. Fowler, Domain Specific Languages, 1st Edition, Addison-Wesley Professional, 2010.

[23] T. Parr, The definitive ANTLR reference: building domain-specific languages, Pragmatic Bookshelf; 1 edition, 2007.

[24] E. P. Klement, E. Pap, R. Mesiar, Triangular norms, Trends in logic, Kluwer Academic Publ. cop., Dordrecht, Boston, London, 2000.

[25] P. Cingolani, J. Alcala-Fdez, jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation, in: FUZZ-IEEE, 2012, pp. 1–8.

[26] P. Cingolani, J. Alcalá-Fdez, jFuzzyLogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming, International Journal of Computational Intelligence Systems 6 (2013) 61–75.

[27] B. N. D. Stefano, On the need of a standard language for designing fuzzy systems, in: G. Acampora, V. Loia, C. Lee, M. Wang (Eds.), On the Power of Fuzzy Markup Language, Vol. 296 of Studies in Fuzziness and Soft Computing, Springer, 2013, pp. 3–15. doi:10.1007/978-3-642-35488-5. URL http://dx.doi.org/10.1007/978-3-642-35488-5

[28] M. Odersky, L. Spoon, B. Venners, Programming in scala, Artima Inc, 2008.

[29] R. R. Redziejowski, From EBNF to PEG, in: Proceedings of the 21th International Workshop on Concurrency, Specification and Programming, Berlin, Germany, September 26-28, 2012, 2012, pp. 324–335.

[30] N. Chomsky, On certain formal properties of grammars, Information and Control 2 (2) (1959) 137 – 167.

[31] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: ACM SIGPLAN Notices, Vol. 39, ACM, 2004, pp. 111–122.

[32] P. H. Nguyen, M. Odersky, Scala benchmarking suite-scala performance regression pinpointing.

[33] G. Hutton, E. Meijer, Monadic parser combinators, School of Computer Science and IT, 1996.

[34] H. Burge William, Recursive programming techniques, Reading Mass: Addison-Wesley, 1975.

[35] P. Wadler, How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages, in: Functional Programming Languages and Computer Architecture, Springer, 1985, pp. 113–128.

[36] G. Hutton, Higher-order functions for parsing, J. Funct. Program. 2 (3) (1992) 323–343.

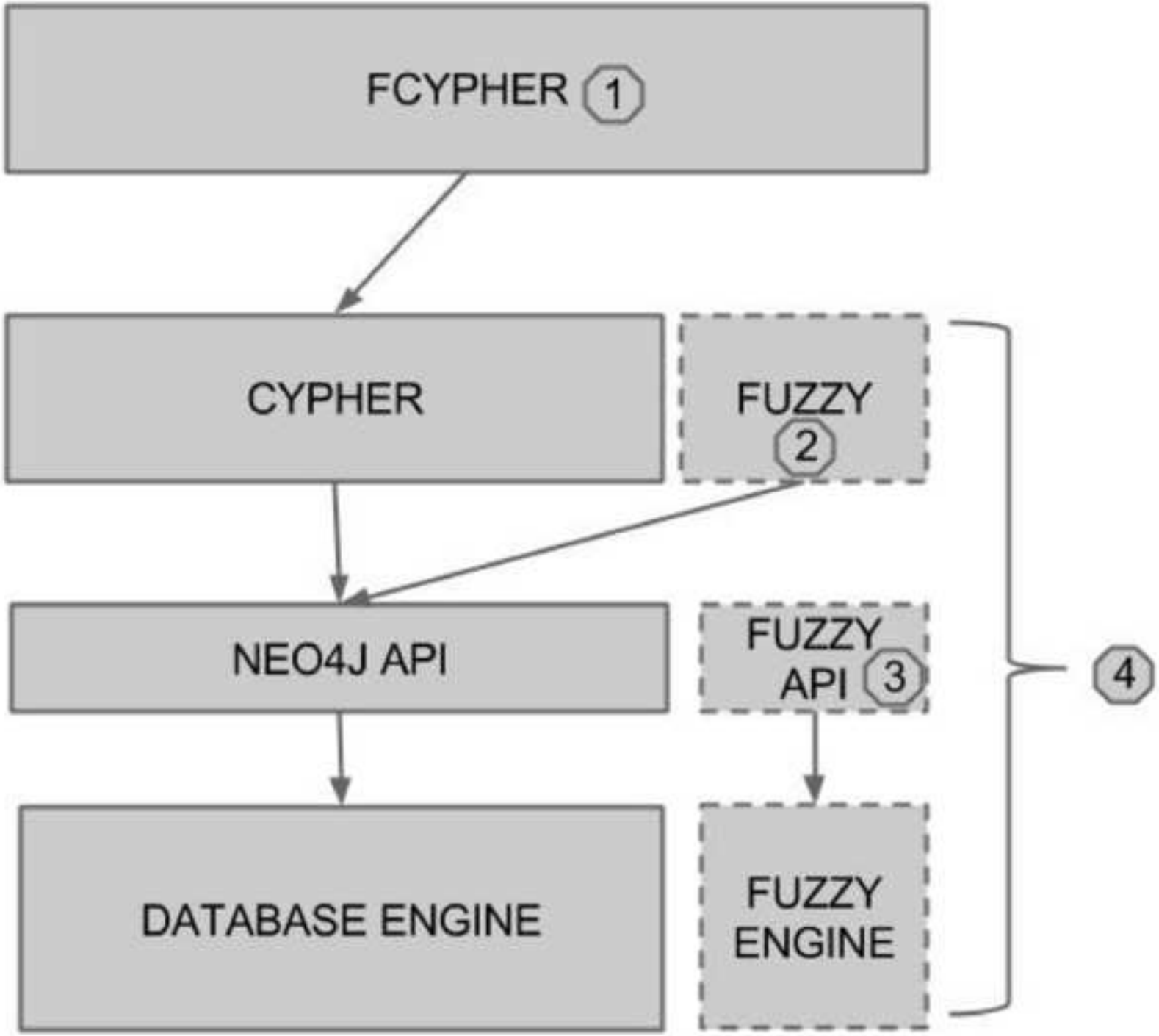
[37] J. Fokker, Functional parsers, in: Advanced functional programming, Springer, 1995, pp. 1–23.

[38] B. Goetz, Java theory and practice: Dynamic compilation and performance measurement, IBM Developer Works (2004) 1–8.

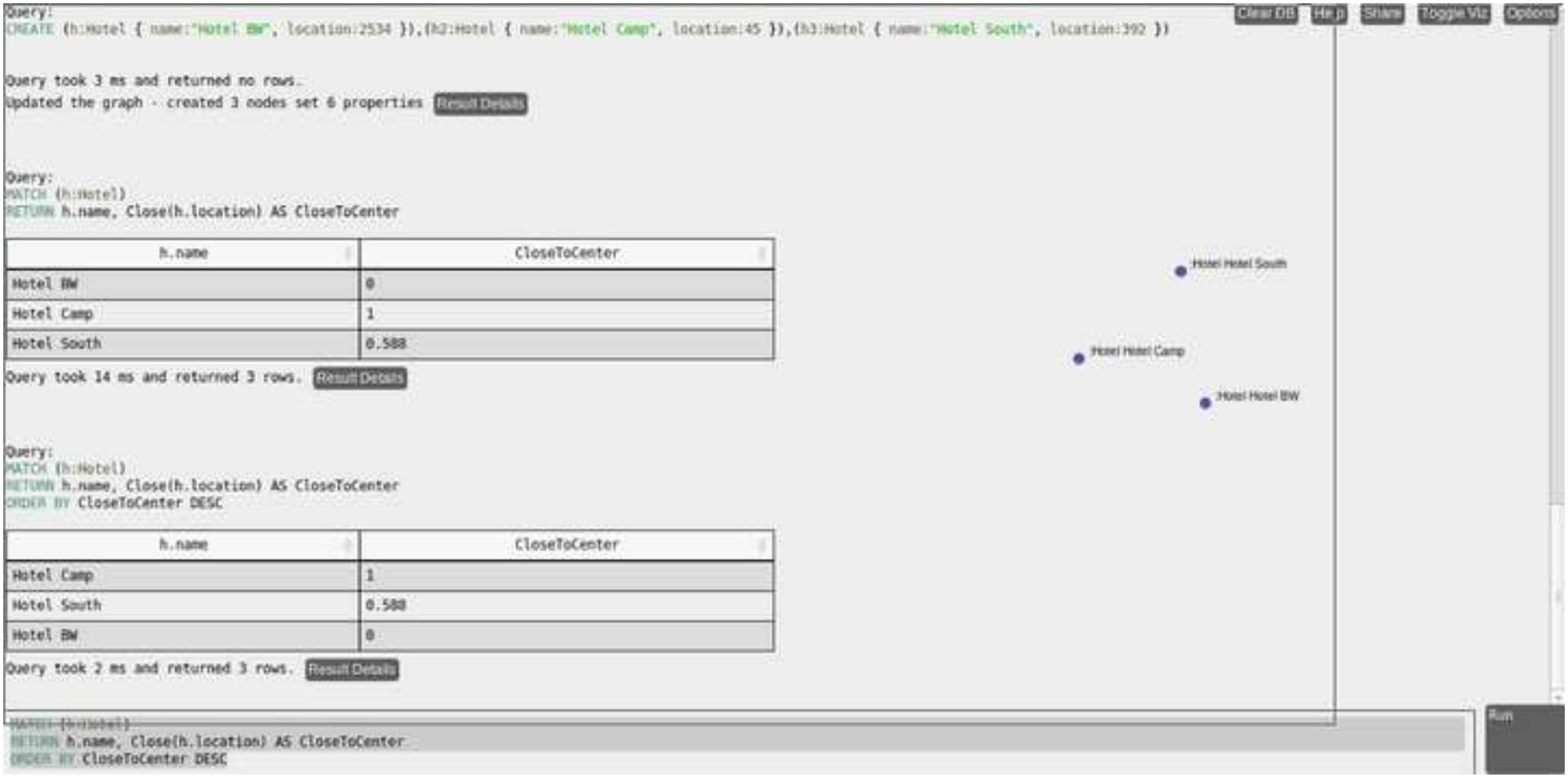
[39] B. Goetz, Java theory and practice: Anatomy of a flawed microbenchmark, IBM developer-Works.

[40] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, ACM SIGPLAN Notices 42 (10) (2007) 57–76.

fcypher-alternatives.png
Click here to download high resolution image

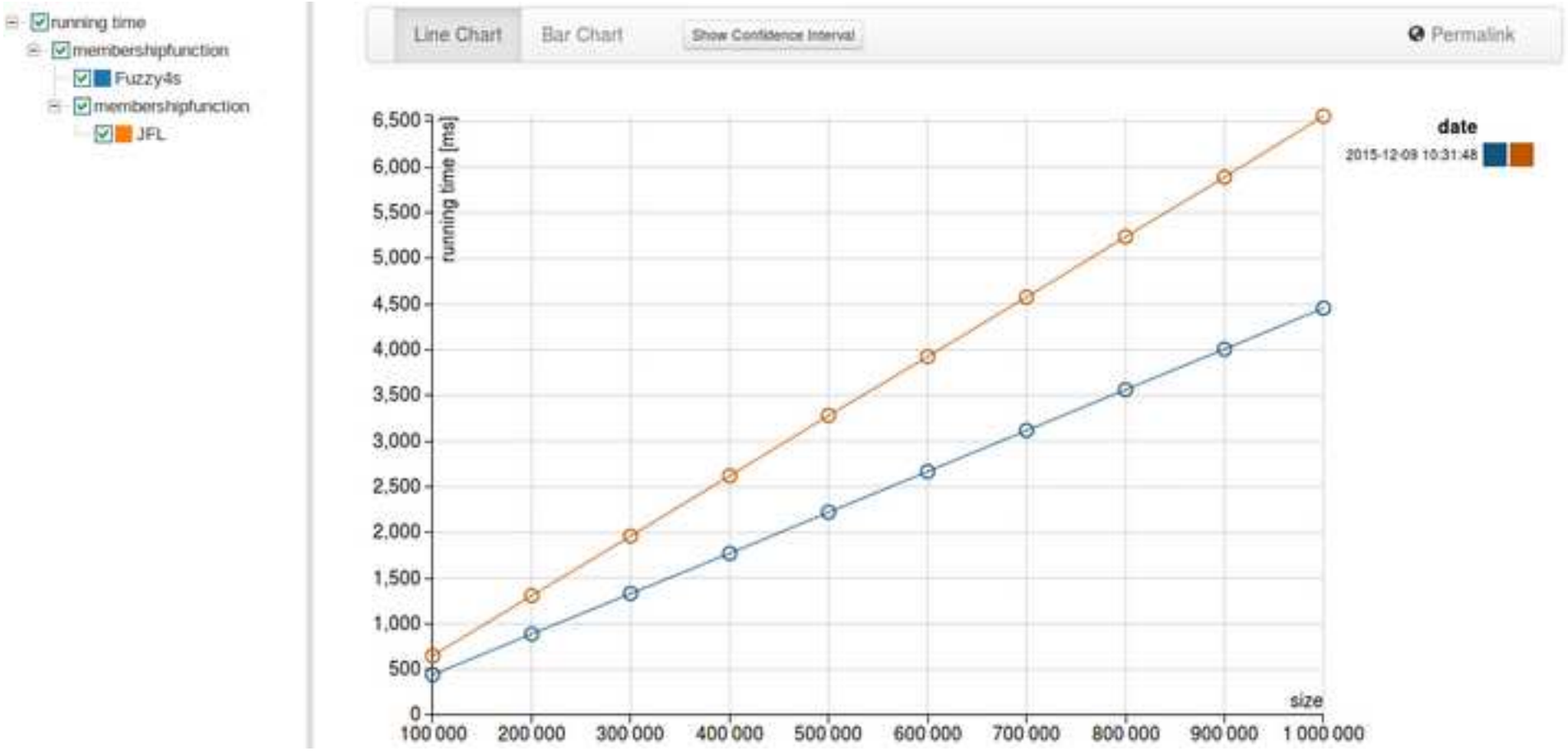


fuzzy-prototype-close.png
Click here to download high resolution image



fuzzy4svsjfl.png

Click here to download high resolution image



gdbms.png
Click here to download high resolution image

19 systems in ranking, November 2015

Rank Nov 2015	Rank Oct 2015	Rank Nov 2014	DBMS	Database Model	Score Nov 2015	Score Oct 2015	Score Nov 2014
1.	1.	1.	Neo4j	Graph DBMS	34.04	+0.63	+9.39
2.	2.	2.	Titan	Graph DBMS	6.06	+0.54	+3.45
3.	3.	3.	OrientDB	Multi-model	5.50	+0.57	+3.48
4.	4.	↑ 7.	ArangoDB	Multi-model	1.61	+0.13	+1.30
5.	5.	↑ 6.	Giraph	Graph DBMS	0.92	-0.00	+0.45
6.	6.	↓ 5.	AllegroGraph	Multi-model	0.91	+0.01	+0.30
7.	7.	↑ 11.	Stardog	Multi-model	0.54	+0.02	+0.41
8.	↑ 9.	↑ 9.	Sqrrl	Multi-model	0.42	-0.01	+0.24
9.	↓ 8.	↓ 8.	InfiniteGraph	Graph DBMS	0.38	-0.05	+0.12
10.	10.	↓ 4.	Sparksee	Graph DBMS	0.29	-0.02	-0.59
11.	11.	↑ 15.	HyperGraphDB	Graph DBMS	0.25	-0.01	+0.22
12.	12.	12.	InfoGrid	Graph DBMS	0.21	+0.00	+0.09
13.	↑ 14.		VelocityGraph	Graph DBMS	0.19	+0.02	
14.	↑ 15.	↑ 16.	GlobalsDB	Multi-model	0.18	+0.02	+0.18
15.	↓ 13.	↓ 13.	FlockDB	Graph DBMS	0.16	-0.02	+0.04
16.	↑ 17.	↓ 10.	GraphDB	Multi-model	0.10	+0.06	-0.06
17.	↓ 16.	↓ 14.	GraphBase	Graph DBMS	0.05	-0.01	+0.02
18.	18.		Blazegraph	Multi-model	0.01	+0.00	
19.	19.	↓ 16.	Amisa Server	Multi-model	0.00	±0.00	±0.00

peg.png
Click here to download high resolution image

Description	PEG notation	Scala notation
Literal string	' '	" "
Literal string	" "	" "
Character class	[ ]	"[ ]".r
Any character	.	".".r
Grouping	(*e*)	(e)
Optional	e?	(e?) or opt(e)
Zero-or-more	e*	(e*) or rep(e)
One-or-more	e+	(e+) or rep1(e)
And-predicate	&e	guard(e)
Not-predicate	!e	not(e)
Sequence	$e_1\, e_2$	e1 ~ e2
Prioritized Choice	e_1 / e_2	e1 | e2

tnorms.png
Click here to download high resolution image

```
$ Match (h:Hotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl...
```

Graph

Rows

hotel.name	distance.name	price.name	minR
h1	far	cheap	1
h2	medium	cheap	1
h5	close	expensive	1
h4	close	midPrice	0.875
h3	medium	midPrice	0.5
h3	medium	cheap	0.33333333333333337
h3	close	midPrice	0.25
h3	close	cheap	0.25
h4	medium	midPrice	0.125
h1	far	midPrice	0
h1	far	expensive	0
h1	close	cheap	0
h1	close	midPrice	0
h1	close	expensive	0
h1	medium	cheap	0
h1	medium	midPrice	0

Returned 45 rows in 81 ms.

tnormswhere.png
Click here to download high resolution image

```
$ Match (h:Hotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl...
```

Graph

Rows

hotel.name	distance.name	price.name	minR
h3	close	cheap	0.25
h1	close	cheap	0
h2	close	cheap	0
h4	close	cheap	0
h5	close	cheap	0

Returned 5 rows in 78 ms.

tnormswheremidprice.png
Click here to download high resolution image

```
$ Match (h1Hotel) WITH h as hotel, h.price as hotelPrice, h.distance as hotelDistance WITH hotel, FuzzyLT("/tmp/distance.fl...
```

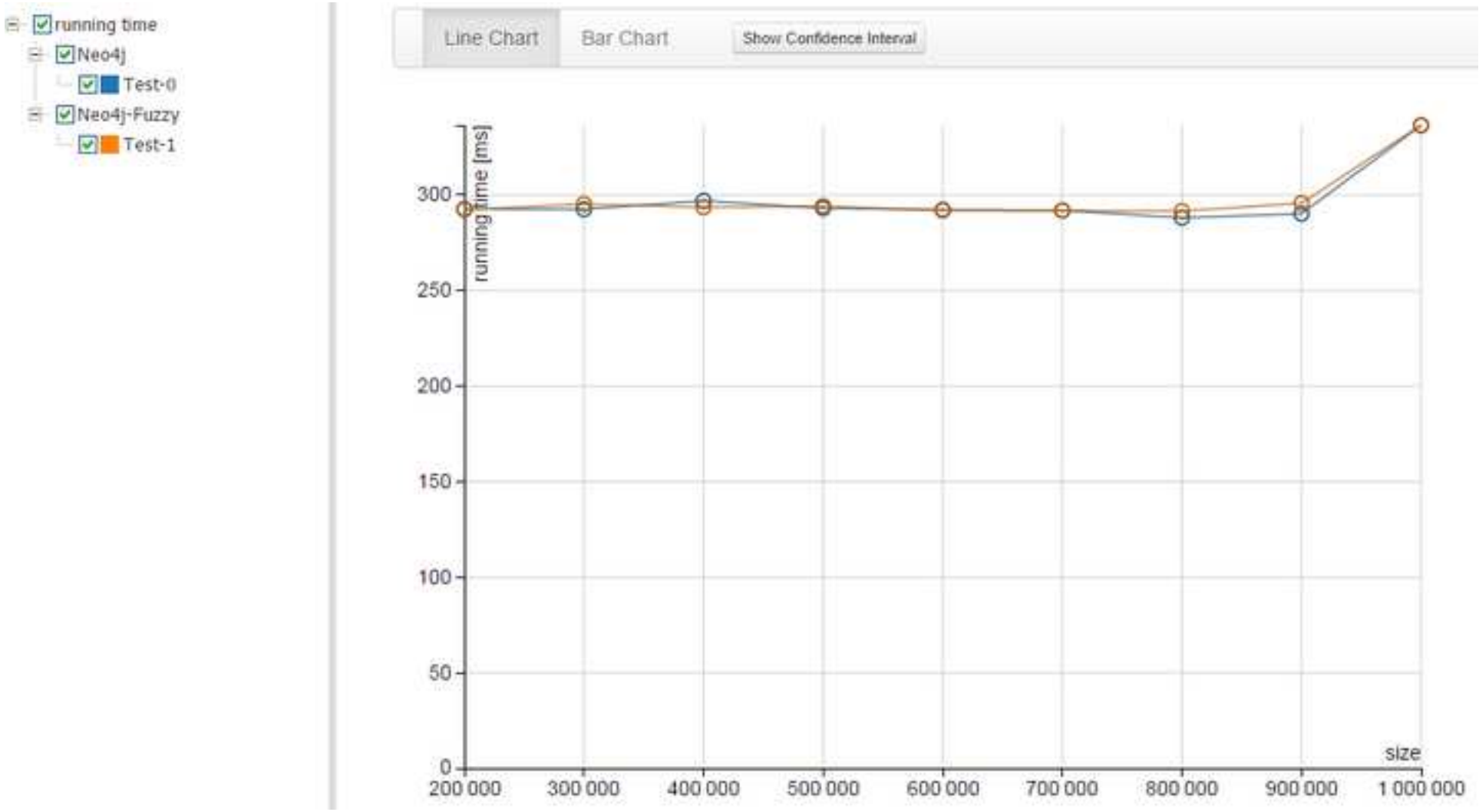
Graph

Rows

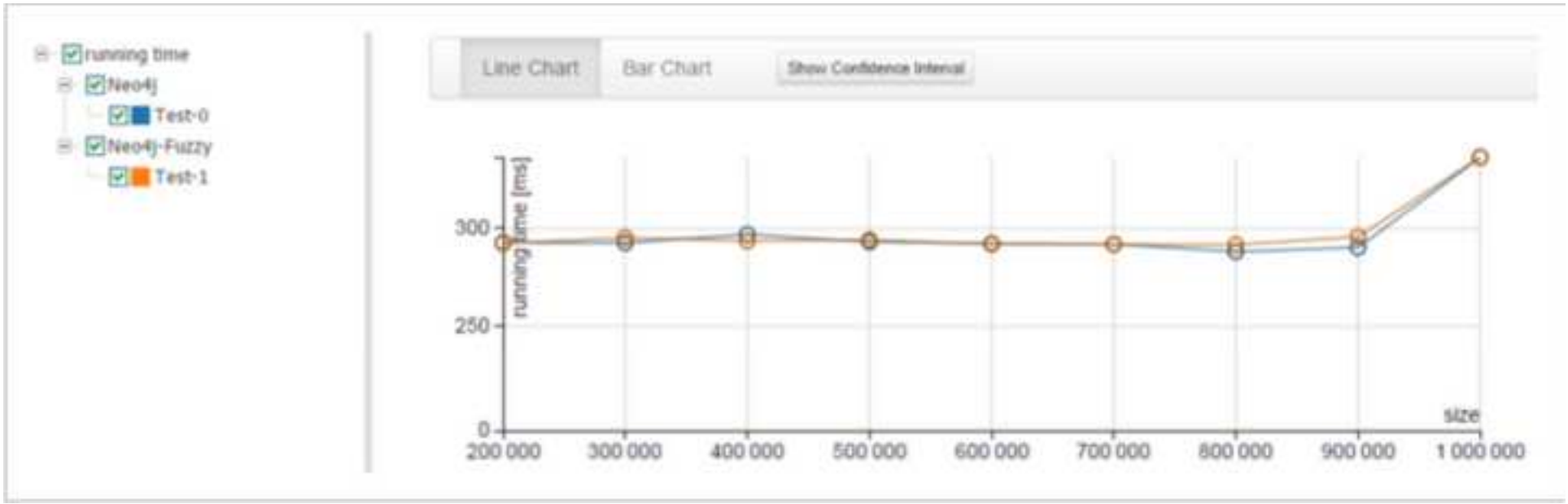
hotel.name	distance.name	price.name	minR
h4	close	midPrice	0.875
h3	close	midPrice	0.25
h1	close	midPrice	0
h2	close	midPrice	0
h5	close	midPrice	0

Returned 5 rows in 1064 ms.

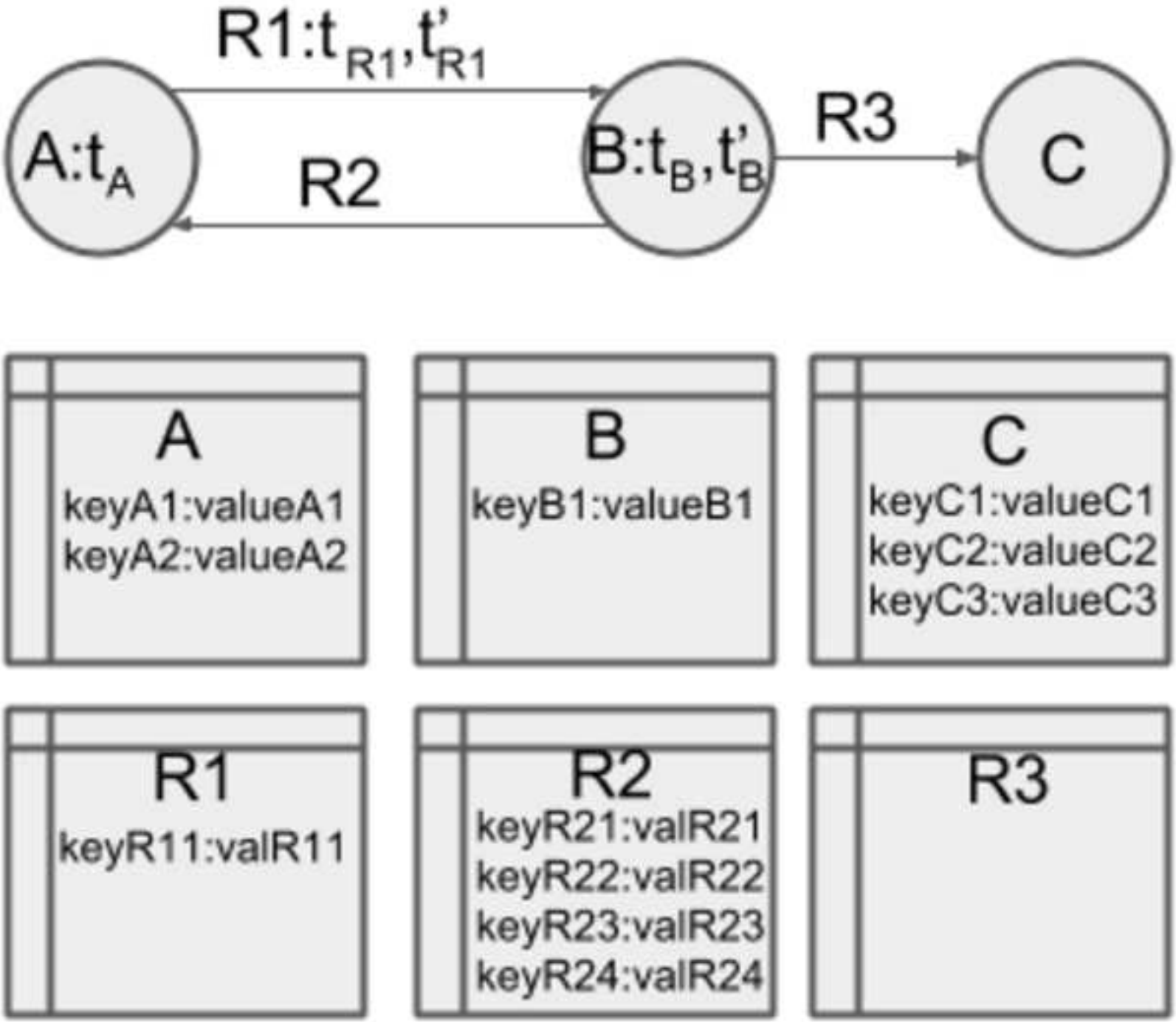
bench-neo.png
Click here to download high resolution image



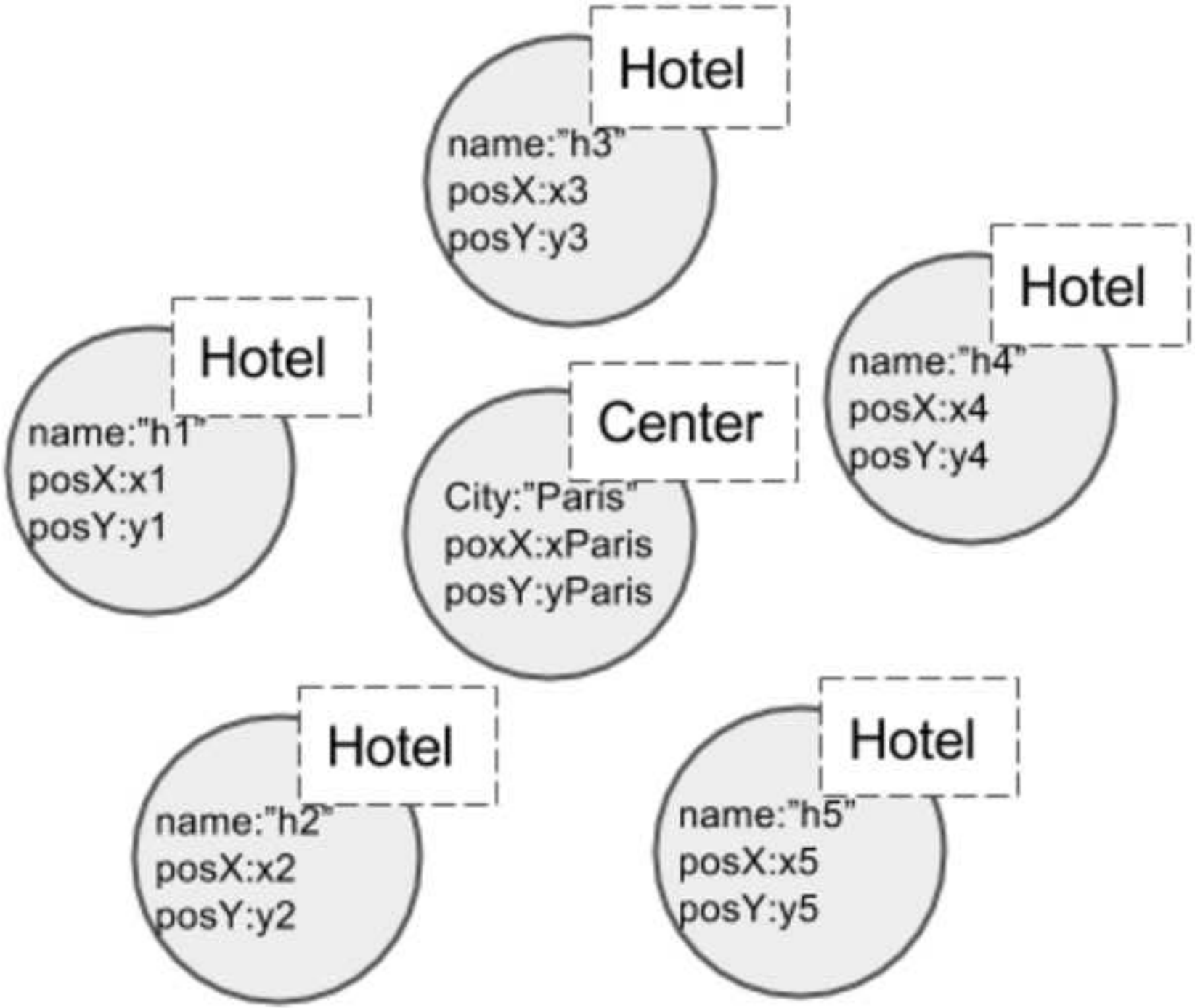
bench-neov2.png
Click here to download high resolution image



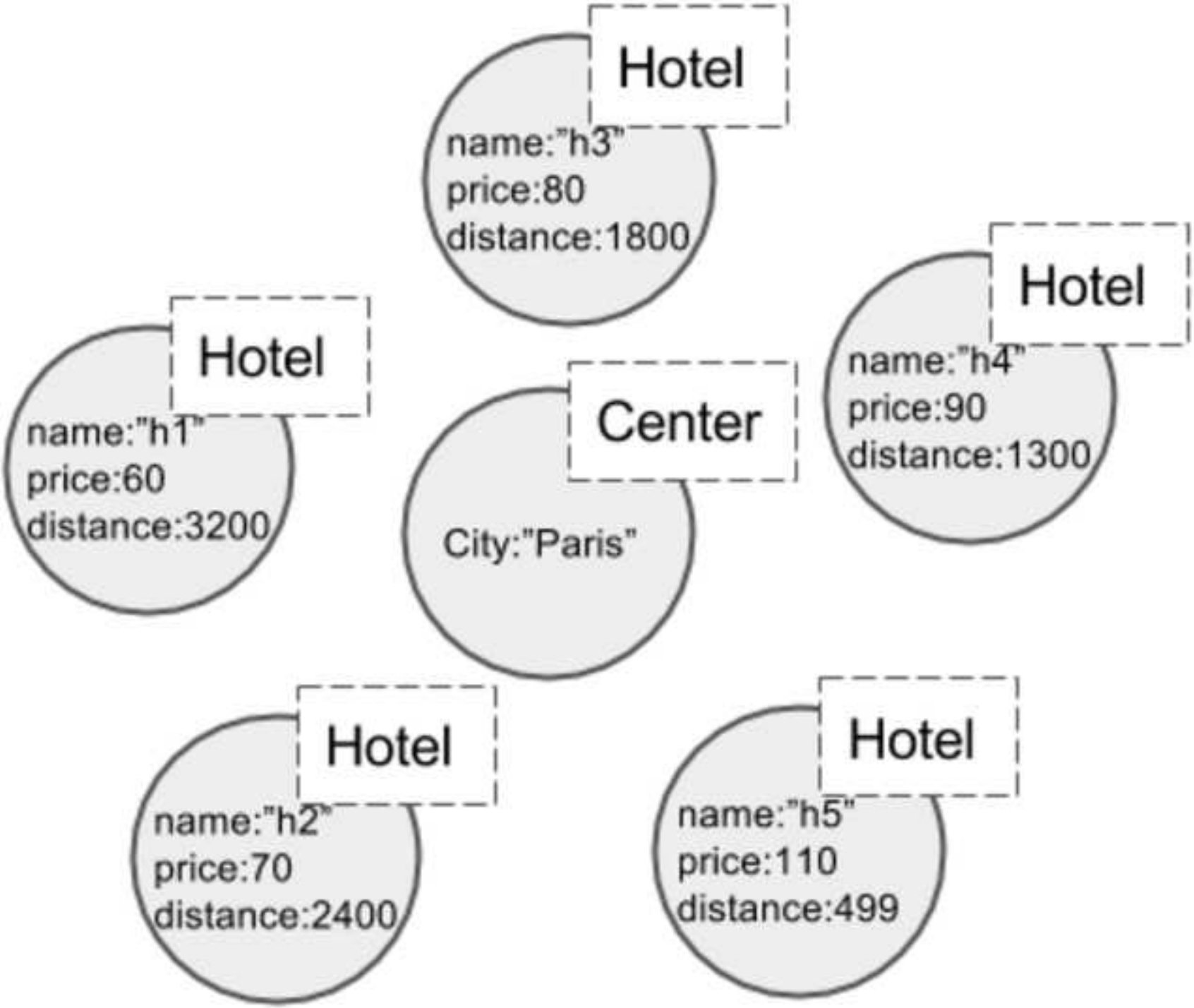
conceptual_example_rels_organisation.png
Click here to download high resolution image



datamodelization.png
Click here to download high resolution image



dataset.png
Click here to download high resolution image

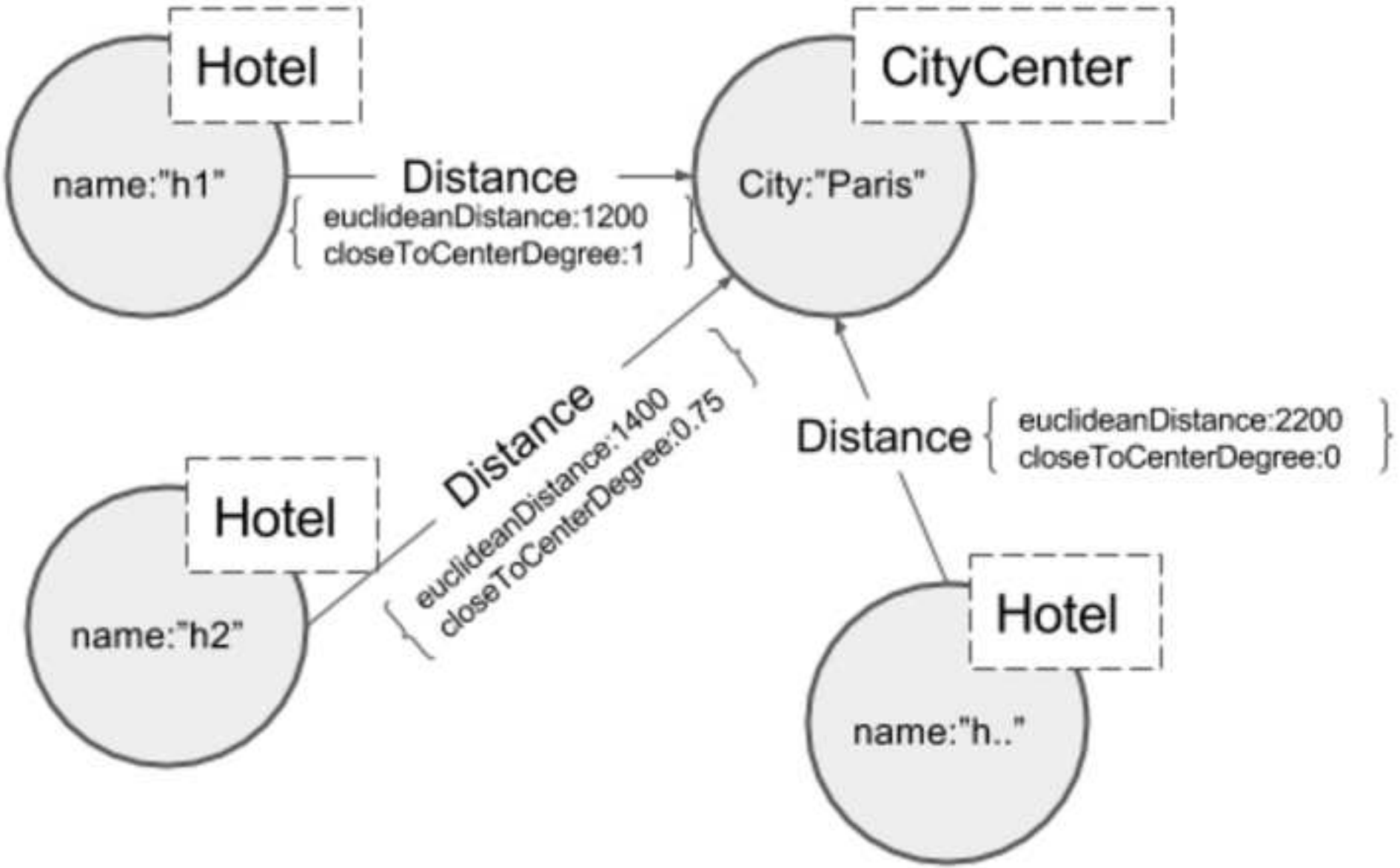


dbengines.png
Click here to download high resolution image

27 systems in ranking, July 2017

Rank Jul 2017	Rank Jun 2017	Rank Jul 2016	DBMS	Database Model	Score Jul 2017	Score Jun 2017	Score Jul 2016
1.	1.	1.	Neo4j	Graph DBMS	38.52	+0.65	+4.83
2.	2.	↑ 4.	Microsoft Azure Cosmos DB	Multi-model	7.71	+1.33	+5.29
3.	3.	↓ 2.	OrientDB	Multi-model	5.57	-0.24	-0.26
4.	4.	↓ 3.	Titan	Graph DBMS	4.93	-0.33	-0.34
5.	5.	↑ 6.	ArangoDB	Multi-model	2.96	-0.12	+1.21
6.	6.	↓ 5.	Virtuoso	Multi-model	1.99	-0.01	-0.31
7.	7.	7.	Giraph	Graph DBMS	1.05	+0.00	+0.16
8.	8.	8.	AllegroGraph	Multi-model	0.61	+0.00	+0.14
9.	9.	9.	Stardog	Multi-model	0.55	+0.01	+0.09
10.	10.	↑ 12.	GraphDB	Multi-model	0.53	+0.01	+0.36
11.	11.	↓ 10.	Sqrrl	Multi-model	0.49	+0.02	+0.24
12.	↑ 16.		Graph Engine	Multi-model	0.36	+0.10	
13.	13.	↓ 11.	InfiniteGraph	Graph DBMS	0.29	-0.01	+0.10
14.	↑ 15.	↑ 18.	Blazegraph	Multi-model	0.29	-0.00	+0.21
15.	↓ 14.		Dgraph	Graph DBMS	0.28	-0.02	
16.	↑ 19.		JanusGraph	Graph DBMS	0.23	+0.11	
17.	↑ 21.	↓ 16.	Sparksee	Graph DBMS	0.16	+0.05	+0.07
18.	↓ 17.	↓ 14.	FlockDB	Graph DBMS	0.16	-0.03	+0.03
19.	↓ 18.	↓ 17.	HyperGraphDB	Graph DBMS	0.15	-0.03	+0.07
20.	20.	↓ 15.	InfoGrid	Graph DBMS	0.13	+0.01	+0.04

euclideanrelationships.png
Click here to download high resolution image



fsparql.png
Click here to download high resolution image

Query	::=	Prologue (*QueryType* SelectQuery | ConstructQuery | DescribeQuery| AskQuery)
QueryType	::=	'#FQ#' | #top-k FQ# with k
Constraint	::=	BracketedExpression | BuiltInCall | FunctionCall | FlexibleExpression
FlexibleExpression	::=	FuzzyTermExpression | FuzzyOperatorExpression
FuzzyTermExpression	::=	(Var ['=','!=','>''>=','<','<='] FuzzyTerm)? [with threshold]
FuzzyOperatorExpression	::=	Var FuzzyOperator NumericLiteral
FuzzyOperator	::=	(Modifier)*FuzzyOperator
FuzzyTerm	::=	FuzzyTerm and FuzzyTerm | FuzzyTerm or FuzzyTerm | not FuzzyTerm | ModifiedFuzzyTerm
ModifiedFuzzyTerm	::=	(Modifier)* ModifiedFuzzyTerm | (Modifier)* SimpleFuzzyTerm

fuzzy4sexamples.png
Click here to download high resolution image

Membership function	Scala code	DSL
PieceWise	PieceWiseMembershipFunction (List(ValuePoint(0,1), ValuePoint(2,1), ValuePoint(4,0)))	(0,1) (2,1) (4,0)
Triangular	TriangularMembershipFunction(0,2.5,5)	trian 0 2.5 5
Trapezoidal	TrapezoidalMembershipFunction(0,2,3,4)	trape 0 2 3 4
Sigmoidal	SigmoidalMembershipFunction(-4, 3)	sigm -4 3
Singleton	SingletonMembershipFunction(SingletonPoint(2))	2
...	...	...

fuzzyltprice.png
Click here to download high resolution image

```
$ match (h:Hotel) return h.name, fuzzyLT("/tmp/price.fl", h.price)
```

Graph

Rows

h.name	fuzzyLT("/tmp/price.fl", h.price)
h1	[
	name cheap degree 1
	,
	name midPrice degree 0
	,
	name expensive degree 0
	]
h2	[

Returned 5 rows in 77 ms.

fuzzyltpricehead.png
Click here to download high resolution image

```
$ match (h:Hotel) with h, h.price as price return h.name, head(fuzzyLT("/tmp/price.fl", price))
```

Graph

Rows

h.name	head(fuzzyLT("/tmp/price.fl", price))
h1	name cheap degree 1
h2	name cheap degree 1
h3	name midPrice degree 0.5
h4	name midPrice degree 1
h5	

Returned 5 rows in 77 ms.

fuzzyltpriceheadname.png
Click here to download high resolution image

```
$ match (h:Hotel) with h, h.price as price return h.name, head(fuzzyLT("/tmp/price.fl", price)).name
```

Graph

Rows

h.name	head(fuzzyLT("/tmp/price.fl", price)).name
h1	cheap
h2	cheap
h3	midPrice
h4	midPrice
h5	expensive

Returned 5 rows in 49 ms.

groupbycat.png
Click here to download high resolution image

```
$ match (h:Hotel) with h, h.price as price With h, head(fuzzyLT("/tmp/price.fl", price)) as priceTerm set h.priceCat = pric...
```

Graph

Rows

h.priceCat	collect(h.name)
cheap	[h1, h2]
expensive	[h5]
midPrice	[h3, h4]

Set 10 properties, returned 3 rows in 719 ms.

mfdistance.png
Click here to download high resolution image

```
$ match (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance return h.name, distance order by distance desc
```

Graph

Rows

h.name	distance
h5	1
h4	0.875
h3	0.25
h1	0
h2	0

Returned 5 rows in 64 ms.

mfdistancesmall.png
Click here to download high resolution image

```
$ match (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance return h.name, distance order by distance desc
```

Graph

Rows

h.name	distance
h5	1
h4	0.875
h3	0.25
h1	0
h2	0

Returned 5 rows in 64 ms.

mfdistancethreshold.png
Click here to download high resolution image

```
$ MATCH (h:Hotel) WITH h, fuzzy("(1200,1) (2000,0)", h.distance) as distance WHERE distance > 0.8 RETURN h.name, distance O...
```

Graph

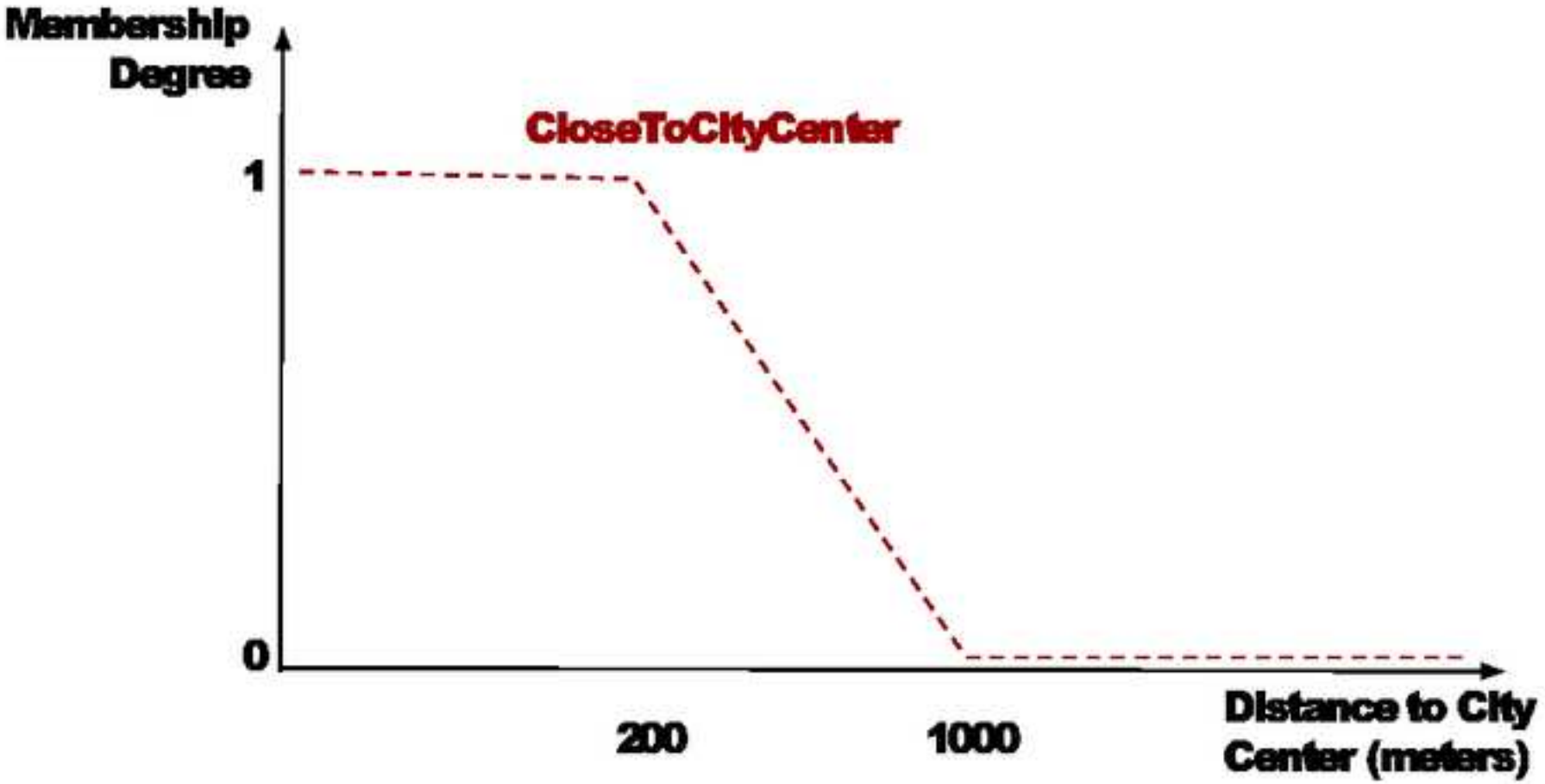
Rows

h.name	distance
h5	1
h4	0.875

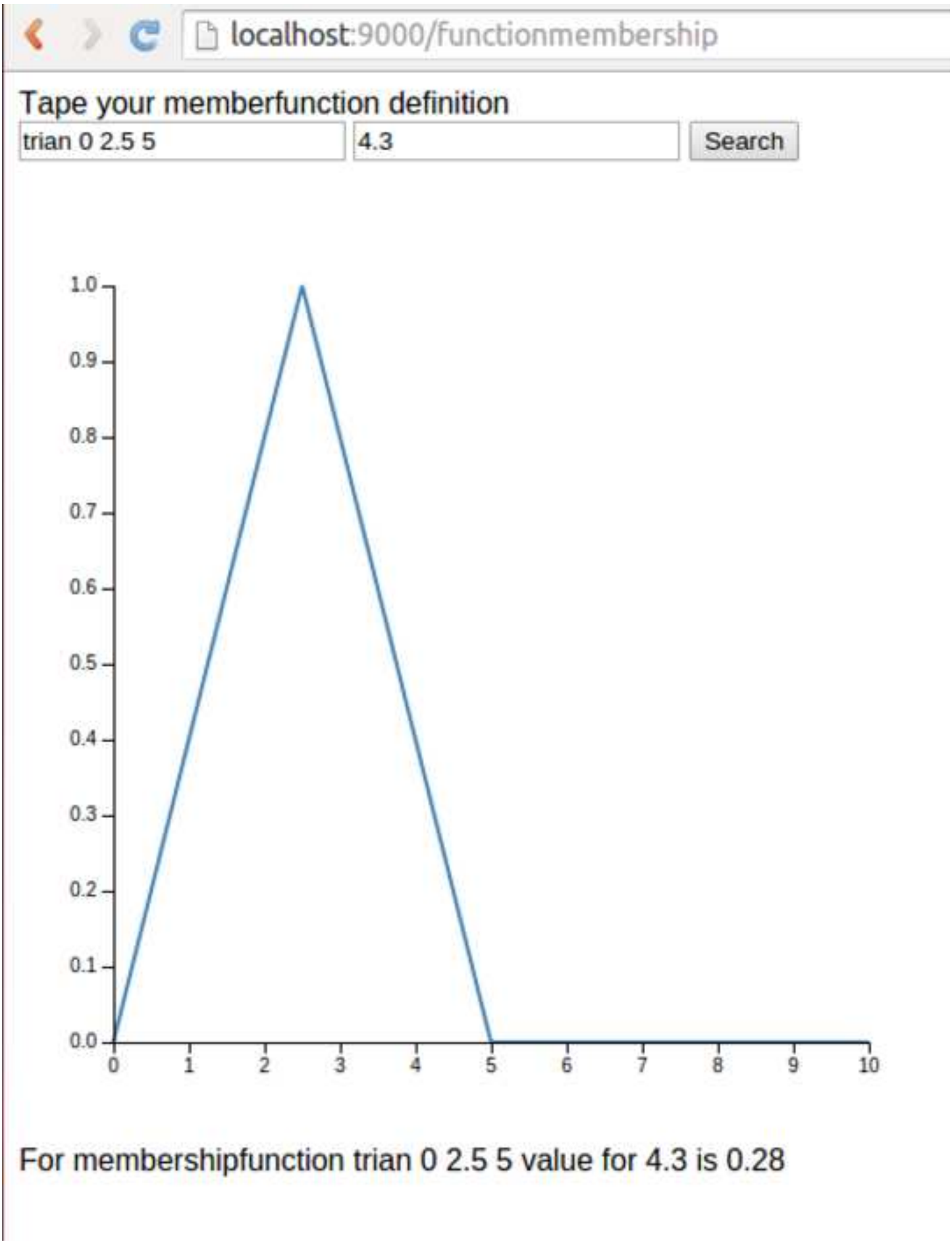
Returned 2 rows in 45 ms.

neo4jfuzzyquery.png
Click here to download high resolution image

```
MATCH (h:Hotel)-[:LOCATED]->(c:City)
RETURN h, CLOSE(h,c) AS ClosenessToCityCenter
ORDER BY ClosenessToCityCenter DESC
```



playfuzzy4s.png
Click here to download high resolution image



asc-aca.tex

Click here to download LaTeX Source Files: asc-aca.tex

biblio-old.tex

Click here to download LaTeX Source Files: biblio-old.tex

biblio.bib

Click here to download LaTeX Source Files: biblio.bib

neo4j-peg.tex

Click here to download LaTeX Source Files: neo4j-PEG.tex

sef.tex

Click here to download LaTeX Source Files: sef.tex